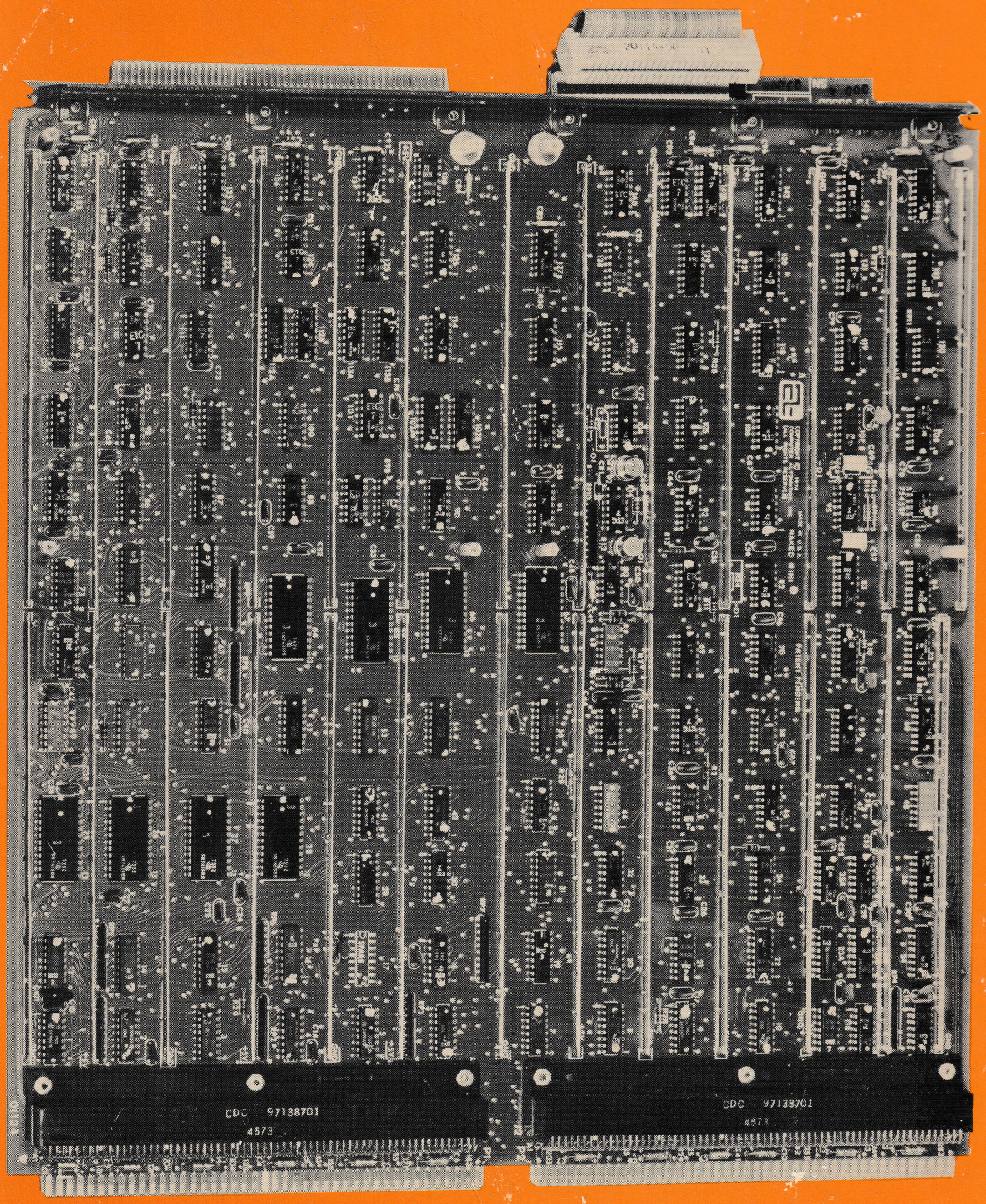
Gunnar Markesjö

Bli bekant med en minidator



entlitteratur

Gunnar Markesjö

Bli bekant med en minidator

Studentlitteratur

© Gunnar Markesjö 1976
Printed in Sweden
Studentlitteratur
Lund 1976
ISBN 91-44-11361-7

Innehåll

0. Inledning 9

1. Räknedosan 13

1.1 Vippor och grindar 16

1.2 Räknedosan 10R 20
1.2.1 Räknedosans blockschema 21
1.2.1.1 Registrens funktion 21
1.2.1.2 Blockschemats uppbyggnad 23
1.2.2 Addition av två tal 23
1.2.3 De fyra räknesätten 25

1.3 Några viktiga begrepp 27

2. En programmerbar kalkylator 29

2.1 Kalkylatorns blockschema 30

2.2 Addition av två tal 31

2.3 Programmering av en enkel addition 32

2.4 Kalkylatorns uppbyggnad 37

2.5 Kalkylatorns minnen 38

2.6 Bandminnet 42
2.6.2 Inspelning och avspelning 44
2.6.3 Ett exempel på användningen av en bandstation 44

2.7 Några viktiga begrepp 49

3. ALPHA LSI 51

3.1 En första presentation 52
3.1.1 Byggblocken 52
3.1.2 Ett förenklat blockschema 55
3.1.2.1 Register 56
3.1.2.2 Minne 56
3.1.2.3 Kontrollpanel 58
3.1.3 Ready console 58
3.1.4 Kontrollpanelen 59
3.1.4.1 Kontrollpanelens register 60

Innehåll

0. Inledning 9

1. Räknedosan 13

1.1 Vippor och grindar 16

1.2 Räknedosan 10R 20
1.2.1 Räknedosans blockschema 21
1.2.1.1 Registrens funktion 21
1.2.1.2 Blockschemats uppbyggnad 23
1.2.2 Addition av två tal 23
1.2.3 De fyra räknesätten 25

1.3 Några viktiga begrepp 27

2. En programmerbar kalkylator 29

2.1 Kalkylatorns blockschema 30

2.2 Addition av två tal 31

2.3 Programmering av en enkel addition 32

2.4 Kalkylatorns uppbyggnad 37

2.5 Kalkylatorns minnen 38

2.6 Bandminnet 42
2.6.2 Inspelning och avspelning 44
2.6.3 Ett exempel på användningen av en bandstation 44

2.7 Några viktiga begrepp 49

3. ALPHA LSI 51

3.1 En första presentation 52
3.1.1 Byggblocken 52
3.1.2 Ett förenklat blockschema 55
3.1.2.1 Register 56
3.1.2.2 Minne 56
3.1.2.3 Kontrollpanel 58
3.1.3 Ready console 58
3.1.4 Kontrollpanelen 59
3.1.4.1 Kontrollpanelens register 60

3.1.4.2 Inskrivning och avläsning 61
3.1.5 Vårt första program 62
3.1.5.1 Instruktionslistan 62
3.1.5.2 Minnet 64
3.1.5.3 Programmet 64
3.1.5.4 Fyra arbetsmoder 68
3.1.5.5 Körningen 69
3.1.6 Vad har vi lärt oss ? 70

3.2 Något om elektroniken 72
3.2.1 Räkneenheten 72
3.2.1.1 Adderaren 72
3.2.1.2 ALU:n 76
3.2.2 Att lagra data 78
3.2.2.1 En latch 80
3.2.2.2 Ett register 82
3.2.2.3 Olika typer av minnen 83
3.2.2.3.1 Ett ROM 83
3.2.2.3.2 Ett RAM 87
3.2.2.3.3 En FILE 88
3.2.3 Att förflytta data 89
3.2.3.1 En buss 90
3.2.3.2 Flera sändare på samma buss 92
3.2.3.3 I/O-bussen 95
3.2.4 Styrenheten 99
3.2.4.1 Blockschemat 100
3.2.4.2 Styrenhetens uppbyggnad 102
3.2.4.2.1 Styrenhet med grindnät 103
3.2.4.2.2 Mikroprogrammerad styrenhet 104
3.2.4.2.3 ALPHA LSI:s styrenhet 106
3.2.5 Exekveringen 111
3.2.5.1 Hämta instruktion 112
3.2.5.2 Verkställ LDA 115
3.2.6 Adressering av en operand 118
3.2.7 Hopp 121
3.2.8 Kraftförsörjningen 122

3.3 Input/Output 126
3.3.1 TTY:n och dess funktioner 126
3.3.2 TTY:n som sändare och mottagare 132
3.3.3 ASCII 133
3.3.4 TTY-interfacet 136
3.3.4.1 Elektroniken 137
3.3.4.2 Programstyrning av TTY:n 141
3.3.4.2.1 Ett enkelt utskriftsprogram 141
3.3.4.2.2 Ett program om ASCII-koden 144
3.3.4.2.3 En enkel läsrutin 147

3.4 Software support 150
3.4.1 Assemblerprogrammering 151
3.4.1.1 AUTOLOAD 152

3.4.1.2 Minnets disposition 153
3.4.1.3 Laddning 154
3.4.1.4 Inkörning av OMEGA och DEBUG 156
3.4.1.5 Vårt första assemblerprogram 159
3.4.1.6 Källkod, objektkod och binärkod 164
3.4.1.7 Från objektremsa till binärremsa 165
3.4.1.8 Felsökning i DEBUG 167
3.4.2 Högnivåspråk 168
3.4.3 Operativsystem 168

3.5 Kassaprogrammet - ett avslutande exempel 171
3.5.1 Önskad funktion 171
3.5.2 Flödesplanen 173
3.5.3 Assemblerprogrammet 176
3.5.4 Några detaljer i assemblerprogrammet 184
3.5.5 Från källkod till kassaprogram 188
3.5.6 Kassaprogrammet i BASIC 188

4. Vidga horisonten

Förord

För utbildning både inom industrin och i skolan har man vid olika tillfällen efterlyst en "datorbok" som på ett enkelt men ändå inte ytligt sätt ger en brygga mellan elektroniken i datorn och datorns programmering. Den här boken är ett försök att fylla detta behov.

När man tränger in i ett nytt område är det ofta enklare att ge sig i kast med en konkret tillämpning än att följa generella linjer i en allmän utveckling. Jag har därför valt att låta framställningen centreras kring en modernt uppbyggd minidator som finns rikligt representerad både inom industri och skola.

För att kunna ge en konkret bild av en specifik minidator och dess funktion krävs ett starkt stöd från dem som yrkesmässigt handhar datorns maskin- och programvara. Detta stöd har jag fått hos Scandia Metrics skickliga medarbetare som med tålamod har hjälpt mig att lösa många problem vid manusarbetet.

Jag vill därför framföra ett varmt tack till alla mina vänner på Scandia Metric. Det är omöjligt att här namnge alla som bidragit, men följande är jag skyldig ett speciellt tack:

Östen Groth (elektroniken i kalkylatorer)
Per-Åke Jansson (elektroniken i ALPHA LSI)
Kurt Eljenmyr (programvaran till ALPHA LSI)
Wally Jonsson (har skrivit kassaprogrammet)
Kurt Sundman (mekaniken och elektroniken i TTY:n)

Christer Eriksson på KTH har bidragit med demonstrationsprogram och synpunkter på bokens uppläggning. I vår undervisning på KTH kommer boken att användas för att exemplifiera digitalkretsarnas användning i datorsystem.

Jag hoppas att det arbete vi alla lagt ner på tillkomsten av denna bok skall bidra till en ökad förståelse för minidatorns funktion och att minska den dimridå, som ofta ligger över gränsområdet mellan maskinvara och programvara.

Sist men inte minst vill jag tacka min hustru Friede som denna heta sommar renskrivit och bearbetat alla manus - även till de tidiga versioner av boken som vi har besparat våra läsare.

St Aspö, oktober 1975

Gunnar Markesjö

Inledning

Mål och förkunskaper

Den här boken har tillkommit för dem som tänker börja använda minidatorer och som dessförinnan vill ha en enkel och konkret beskrivning hur en minidator fungerar. För att konkretisera framställningen ska vi fokusera intresset kring en typisk minidator som heter ALPHA LSI, men det mesta som sägs kommer att vara tillämpbart på alla typer av medelstora minidatorer.

Man talar ofta om maskinvara och programvara (hardware och software). Vi ska ägna oss mest åt maskinvaran, dvs datorns uppbyggnad och funktion. Vår bok är avsedd för de blivande användare av minidatorer som av tekniskt intresse gärna vill lära känna sin dator litet mer från insidan.

Vilka frågor är det då vi ska försöka finna svaret på? Ja, det är inte vad man kan använda en minidator till - det är en omöjlig fråga att besvara. En dator kan fylla en rad olika funktioner, alltifrån att vara en statuspryl till att vara ett redskap med vars hjälp man kan förtjäna sitt dagliga bröd.

Våra frågeställningar är framför allt hur en minidator fungerar, hur man kör igång den och hur man på enklaste sätt kan kommunicera med den. Har vi fått dessa frågor besvarade så har vi också slagit hål på den mur av mystik som elektroniska prylar ofta omges med.

Vi förutsätter att läsaren är bekant med det binära talsystemet och har inhämtat elementära kunskaper om dioden och transistorn på annat håll. (Det finns ett flertal lättlästa läroböcker om elektronik för nybörjare. Referenser ges i slutet av boken).

Innehållet

För att principerna inte ska försvinna i apparatens komplexitet ska vi närma oss vår minidator i tre etapper.

Vi ska först bekanta oss med några viktiga byggblock (exempelvis register och adderare) och några vanliga begrepp (exempelvis data och instruktioner). Detta gör vi i anslutning till en enkel räknedosa.

Vi ska sedan bekanta oss med en programmerbar kalkylator. Där kommer instruktionerna att ställas samman till program och där kommer vi även i kontakt med minnen av olika slag.

Med räknedosan och kalkylatorn som grund ska vi sedan ge oss i kast med vår minidator, ALPHA LSI. Vi kommer då att känna igen många av de byggblock och funktionsprinciper vi tidigare mött i räknedosan och kalkylatorn

Minidatorn ALPHA LSI skall vi sedan angripa i fyra steg. Först ska vi ge en översiktlig beskrivning och köra ett enkelt demonstrationsprogram från kontrollpanelen.

Som andra steg ska vi ge oss in i den elektroniska uppbyggnaden och se efter vad som i princip döljer sig inuti alla de svarta små platskapslarna på kretskorten och hur de samarbetar för att fylla sina uppgifter. Det är alltså här bokens tyngdpunkt ligger.

Som tredje steg ska vi se hur datorn kommunicerar med omgivningen - i vårt fall är det en enkel teletype som utgör in-utenhet.

Som fjärde steg ska vi bekanta oss med ett maskinnära program som kallas assemblerprogram och några programvaruhjälpmedel som brukar användas.

Som avslutning ska vi både köra och beskriva ett demonstrationsprogram som vi har döpt till Kassaprogrammet.

En dator - vad är det?

Innan vi ger oss i kast med räknedosan ska vi kanske först reda upp begreppen en smula. Vad är egentligen en dator (computer) - dvs det som tidigare kallades datamaskin?

SEK ger följande definition[x)] (SEN 011601):

dator, datamaskin
E computer
F calculateur
databehandlare som utan mänskligt ingripande kan utföra omfattande beräkningar med stort antal aritmetiska eller logiska operationer

En enkel räknedosa benämnes enligt SEN på följande sätt:

räknemaskin, räknare
E calculator
F calculatrice
databehandlare som är särskilt lämpad för aritmetiska operationer och som kräver täta ingripanden av en mänsklig operatör

Enligt SEN (och IEC) är tydligen skillnaden mellan begreppen räknare och dator, att räknaren kräver upprepade manuella operationer medan datorn kan stå och räkna en lång stund alldeles för sig själv. Det är alltså inga skarpa gränser mellan de två begreppen räknare och dator.

Många lägger en speciell betydelse i begreppet dator. De menar att det specifika med en dator är att den har sitt "arbetsprogram" lagrat i sitt minne. Vill vi att datorn ska byta arbetsuppgift så behöver vi bara byta ut programmet i minnet - dvs läsa in ett nytt program i minnet. Räknaren har däremot en gång för alla konstruerats med vissa fast inbyggda program och den är därmed låst till motsvarande arbetsuppgifter.

---

x SEK = Svenska Elektriska Kommissionen
SEN = Svenska Elektrotekniska Normer
IEC = International Electrotechnical Commission

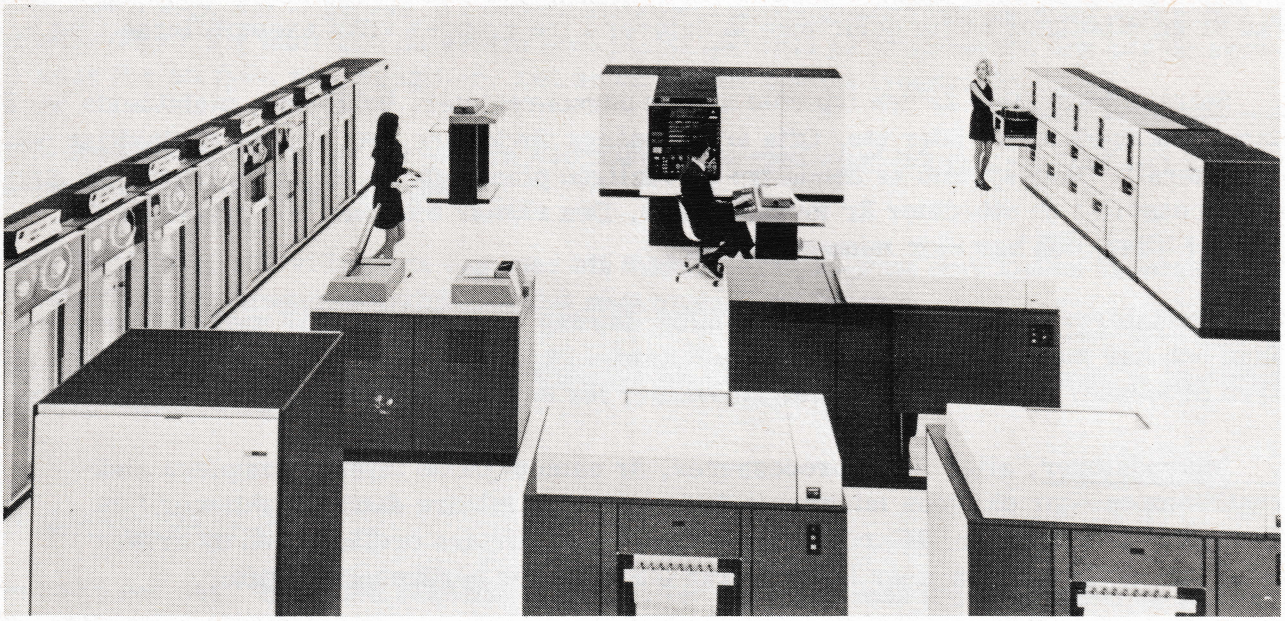
Vad är en minidator? Ofta delar man in datorer i tre grupper vilka exemplifieras i fig 0.1.

1. Stora datorsystem av den typ som sköter personregister, bokföring, omfattande beräkningar exempelvis av väder etc. Ofta är det många användare som samtidigt utnyttjar denna datakraft. Datorsystem av denna typ är vanligen placerade på speciella centraler. Själva processorn i en stordator är mycket snabb. Den räknar ofta med hög precision (32 eller 64 bitar) och har stor minneskapacitet.

2. Minidatorer brukar man kalla de mindre datorer som ofta arbetar med 16 bitars ordlängd och har mer begränsat minne (ofta 4 kord - 64 kord). Minidatorer används oftast att styra processer eller som beräkningsmaskiner för enstaka eller mindre antal användare.

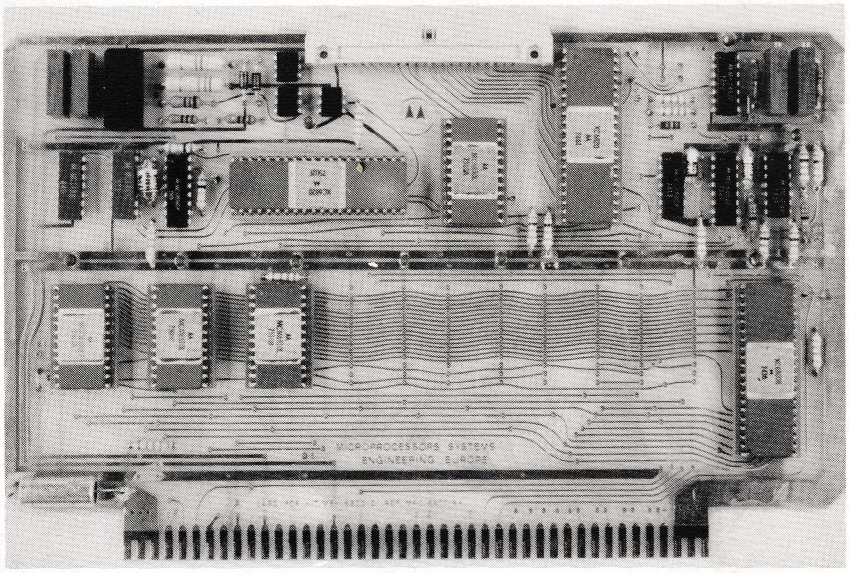
3. Mikrodatorer, eller mikroprocessorer, är namnet på de mycket komplexa elektronikkomponenter där man integrerat de flesta av en mindre dators funktioner i en eller ett fåtal kristaller. Mikrodatorer arbetar med kortare ordlängd och de kräver en hel del anpassningskretsar innan de kan utnyttjas för beräkningsuppgifter.

Vi ska inte fördjupa oss mer i klassificering av datorer. Vår uppgift är ju enklare - vi ska studera ett konkret exempel, minidatorn ALPHA LSI.



a) IBM 370. Exempel på ett stort datorsystem.

b) ALPHA LS I. En minidator.



c) M 6800. Mikrodator, bestående av processor (CPU) och minnen (ROM och RAM).

Fig 0.1 Små och stora datorer.

1. Räknedosan

Vi har nyligen upplevt en omställning på räknemaskinmarknaden från de tidigare mekaniska räknemaskinerna till dagens elektroniska. Det är inte enbart inom handel och industri som de elektroniska kalkylatorerna gjort sitt segertåg, räknedosor är i dag nästan var mans egendom. I skolan ersätter räknedosan alltmer de tidigare räknestickorna.

Orsaken till denna revolution är de nya halvledarkomponenterna och digitaltekniken. Fig 1.1

På 50-talet byggdes de första datorerna med transistorer. Det gick åt många transistorer och tekniken var för dyrbar för att kunna konkurrera med mekanisk uppbyggnad av bordskalkylatorer.

På 60-talet kom de integrerade kretsarna, dvs kristaller som innehöll bortåt ett hundratal transistorer, hopkopplade till enkla funktionsenheter. Därmed blev det ekonomiskt genomförbart att bygga små och billiga datorer, s k minidatorer, för speciella arbetsuppgifter.

Vid början av 70-talet hade halvledartekniken utvecklats så långt att nästan alla kretsar för en enkel bordskalkylator kunde integreras i en enda kristall. Därmed blev elektroniska kalkylatorer både prismässigt och tekniskt en bättre produkt än de tidigare mekaniska kalkylatorerna. Högintegrerade kretsar av denna typ brukar kallas för LSI-kretsar (large scale integration) eftersom så många transistorer integreras i samma kristall.

Att LSI-tekniken blir ekonomiskt förmånlig beror på att ett stort antal kalkylatorkretsar kan tillverkas samtidigt i en och samma kristall. Fig 1.2 visar en kiselskiva som genomgått ett antal processer på halvledarfabriken och är färdig att skäras upp till mer än hundra kalkylatorkristaller. Vi ser också en närbild på en kalkylatorkrets. Kalkylatorkretsen utgör huvuddelen av den elektronik som behövs för en enkel räknedosa.

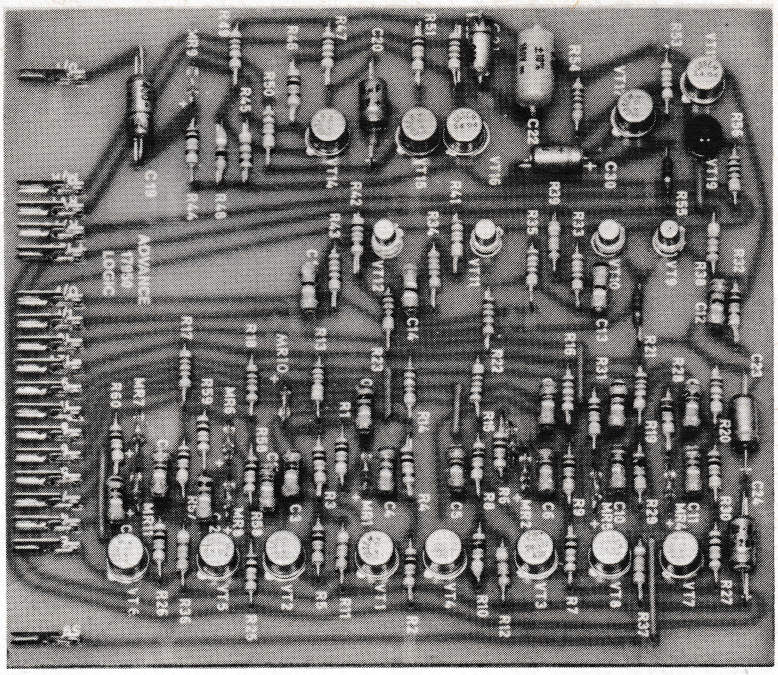
Massproduktionen av kalkylatorkristaller gör att dessa kan säljas för några kronor per kristall (i stora mängder).

emitter

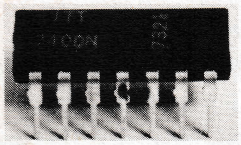
kollektor

bas

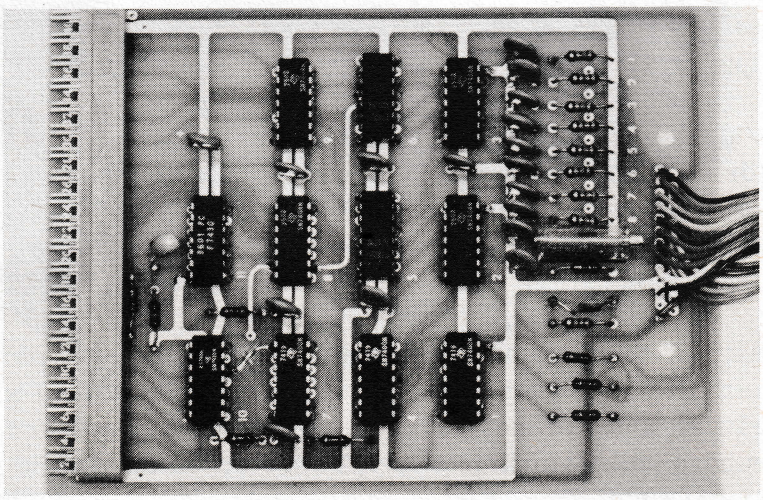
Transistor
1950-talet



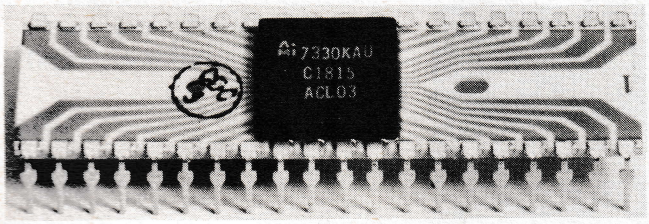
Kretskort med »diskret uppbyggnad»



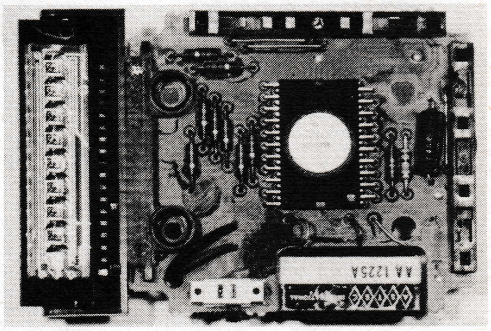
IC-krets
1960-talet



Kretskort med »integrerad uppbyggnad» .

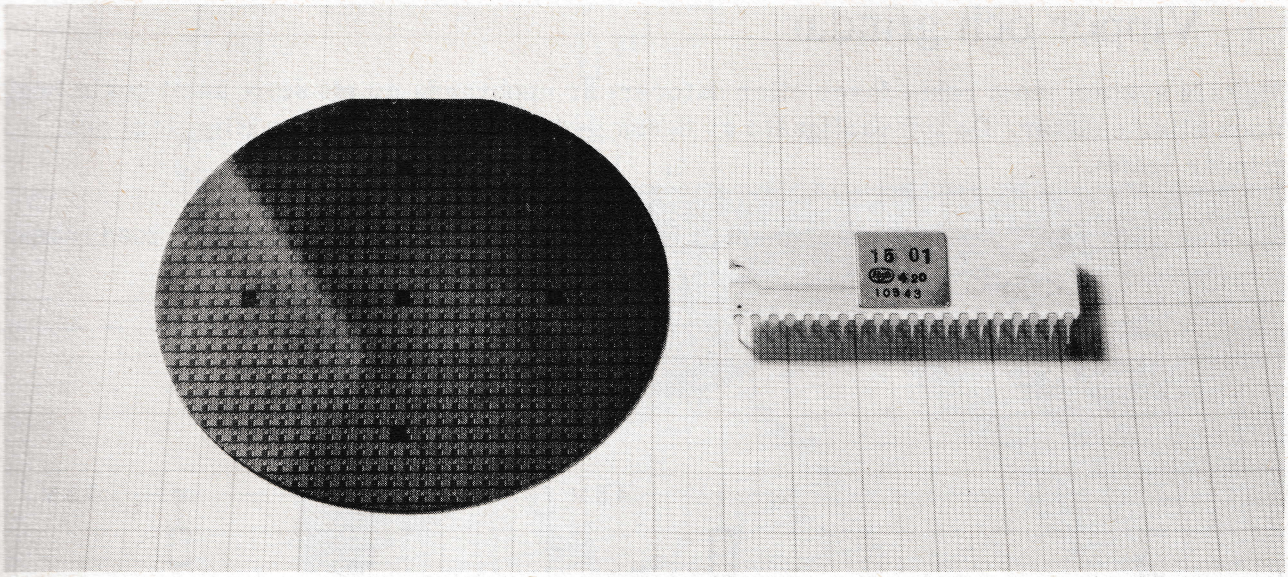


LSI-krets
1970-talet



Kretskort för kalkylator

Fig 1.1 Tre steg i utvecklingen av halvledartekniken.



Skiva innehållande hundratalet kretsar

En kalkylatorkrets monterad i sin kapsel

Fig 1.2 Processad kristallskiva och en av kretsarna i sin kapsel.

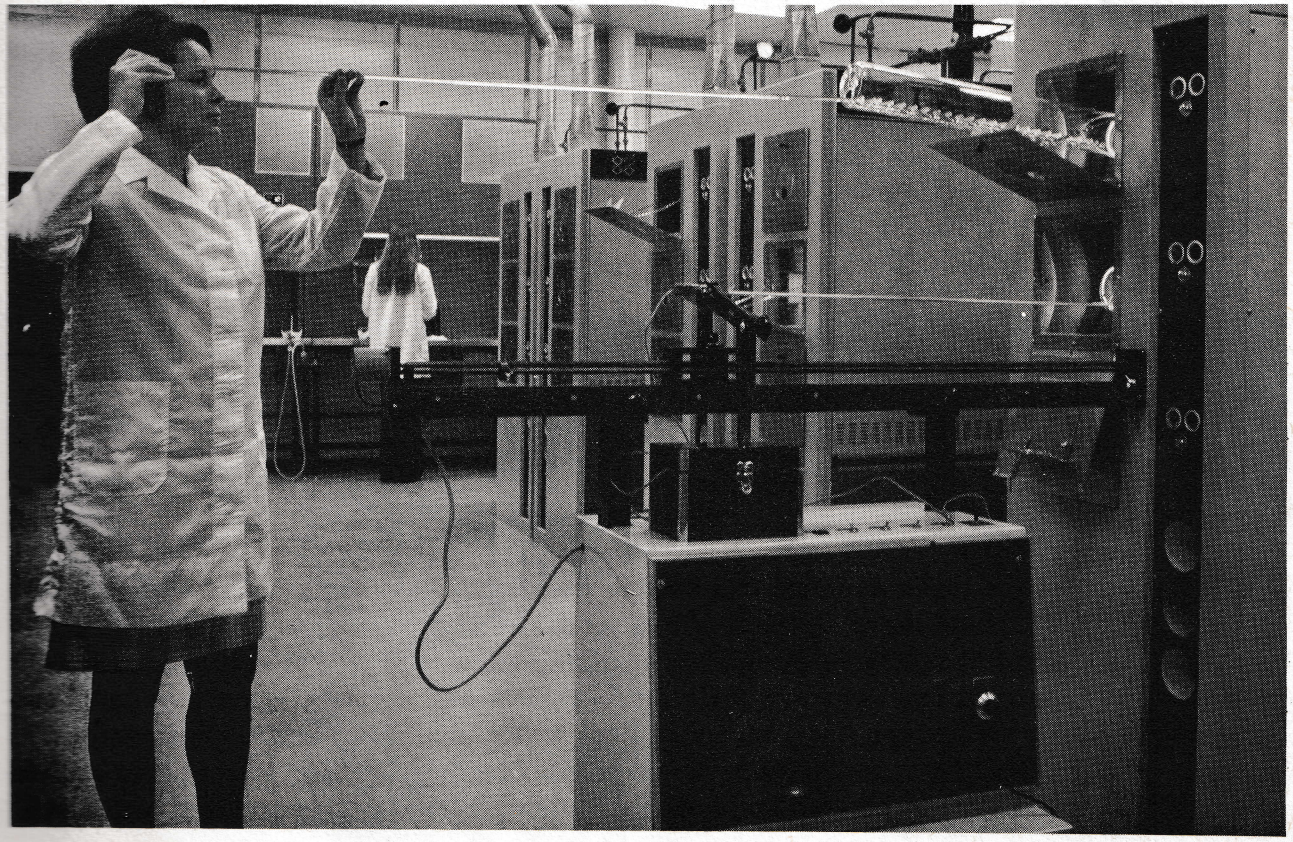


Fig 1.3 Interiör från halvledarfabrik. (RIFA, Stockholm)

1.1 Vippor och grindar

Digitala system (som räknedosor eller datorer) är uppbyggda av ett stort antal enkla byggblock eller enheter. De två vanligaste av dessa byggblock kallas vippor (flip flop) och grindar (gate).

Fig 1.4 visar som exempel kretsscheman för en vippa och en grind uppbyggda med transistorer och dioder.

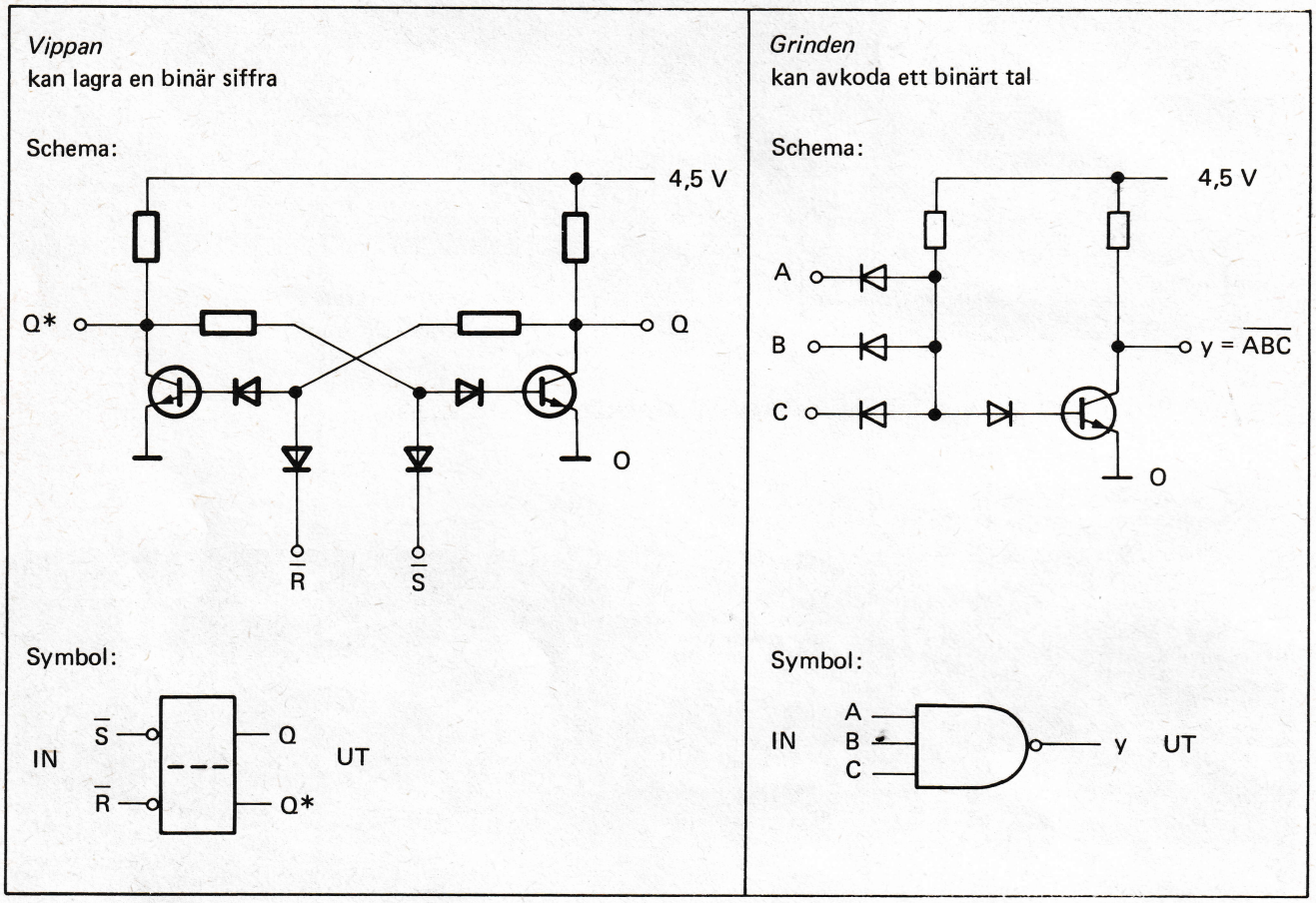


Fig 1.4 Exempel på en enkel vippa och en enkel grind.

Vippan har två stabila lägen som den kan ställas i med hjälp av två ingångar ($\bar{R}$ och $\bar{S}$ i fig 1.4). Den kan därmed lagra en binär siffra (noll eller ett). Vippans innehåll kan man avläsa på utgångarna (Q och Q*). Har Q hög spänning (ca 4 V i fig 1.4) står vippan i läge ett. Vi ska inte gå in på funktionen i detalj här.[x] Ett bra sätt att övertyga sig om att kretsen fungerar är att koppla upp den - på fig 1.5 har vi lött upp den på en bit spånplatta. (För att vi bättre skall se vippans ställning har vi kopplat in lysdioder i serie med kollektormotstånden i fig 1.5. En lysdiod lyser när strömmen genom den överstiger någon mA.)

---

[x] En beskrivning återfinns i Markesjö, Elektronik, Inledande kurs B, Läromedelsförlagen, Stockholm 1970.

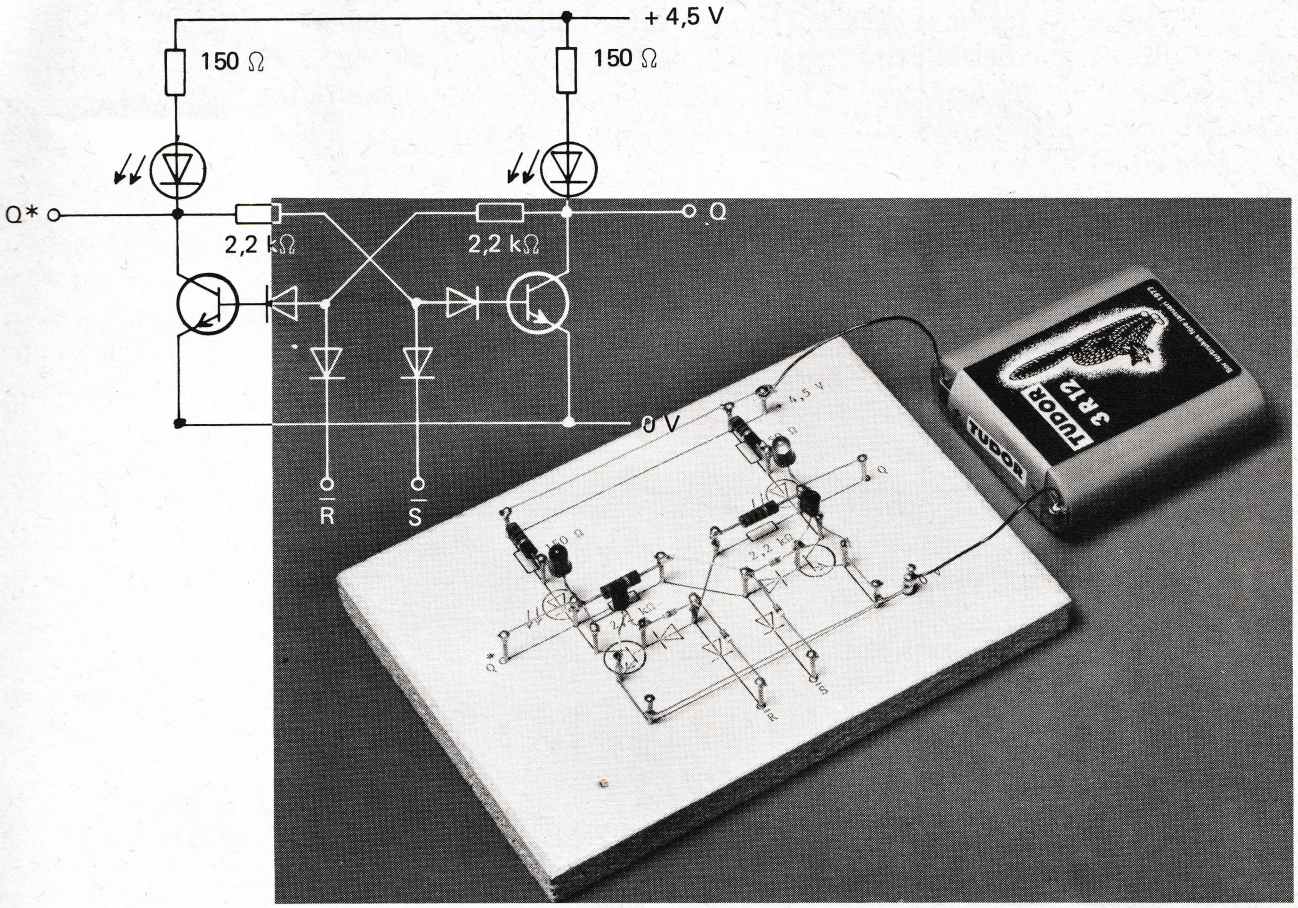


Fig 1.5 Vippan i fig 1.4 uppkopplad på "spikbräda".

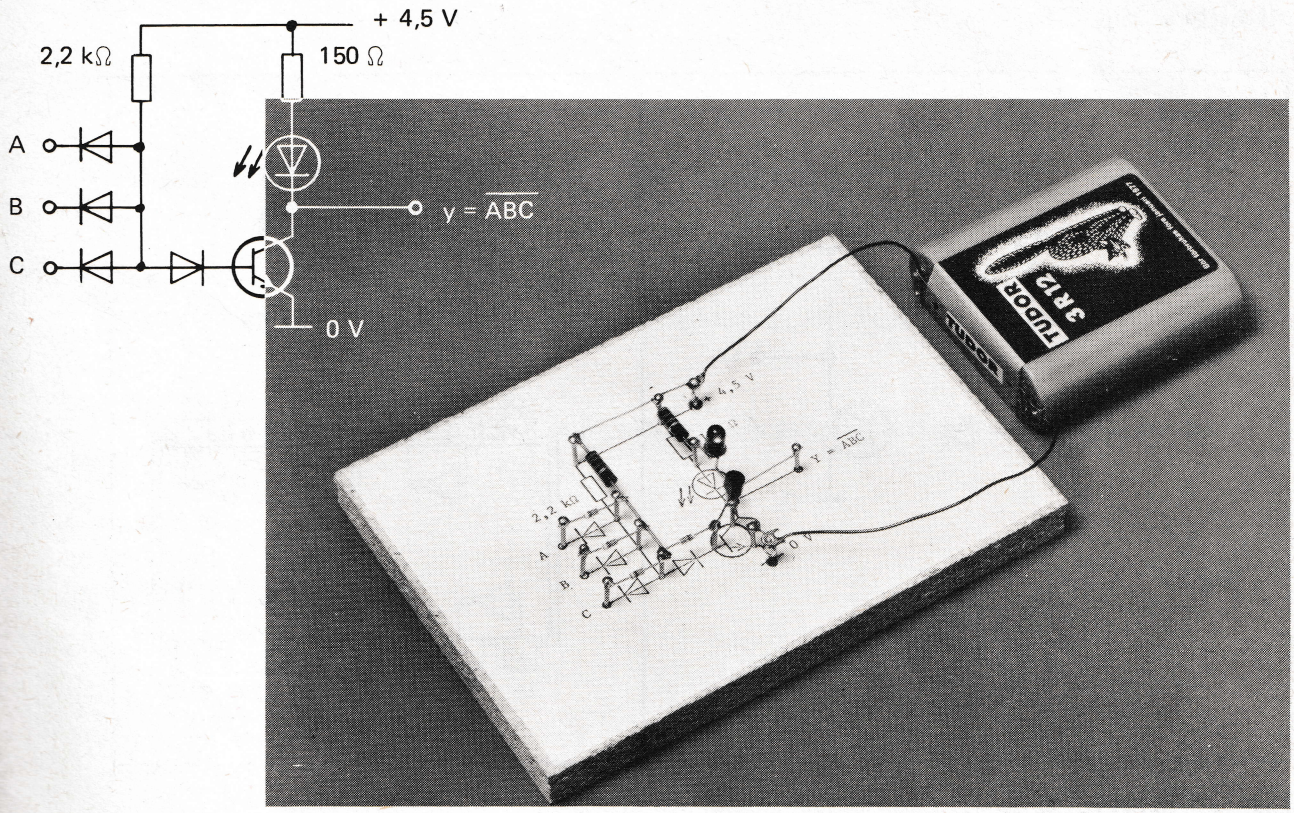


Fig 1.6 Grinden i fig 1.4 uppkopplad på "spikbräda".

En grind utför en logisk funktion. Grinden i fig 1.4b får låg utspänning enbart för en av de 8 möjliga kombinationer på spänningarna (hög eller låg) till ingångarna A, B och C. Enbart när samtliga ingångar är höga blir utgången låg. En grind (eller kombinationskrets) kan därmed fungera som en avkodare som reagerar enbart för en viss kombination av insignaler.

I fig 1.6 har vi kopplat upp grinden från fig 1.4b med transistorer. Koppla gärna upp en grind och prova igenom samtliga kombinationer av insignaler (öppen ingång fungerar som hög). Man frågar sig kanske varför kretsen kallas grind. Orsaken är att den kan användas som en grind som stoppar eller släpper fram en digital signal. Fig 1.7 visar en grind med två ingångar, varav den ena fungerar som signalingång och den andra som kontrollingång (öppnar eller stänger).

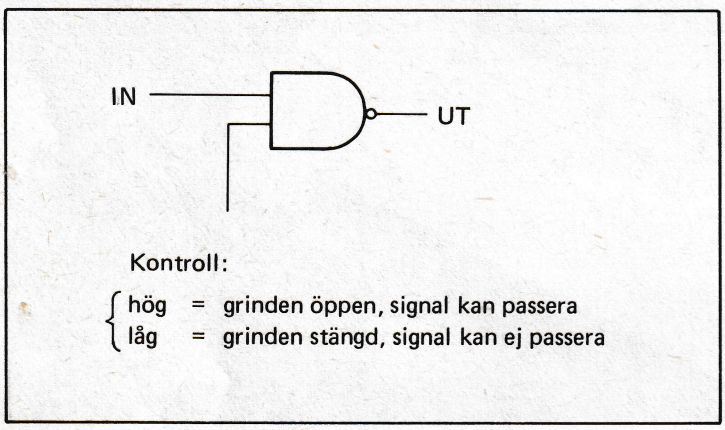


Fig 1.7 En grind för digitala signaler.

Ofta behöver man invertera en signal, dvs byta etta mot nolla eller hög mot låg. En grind enligt fig 1.4b men med enbart en ingång ger denna egenskap. Invertering är alltså enkel att utföra, fig 1.8.

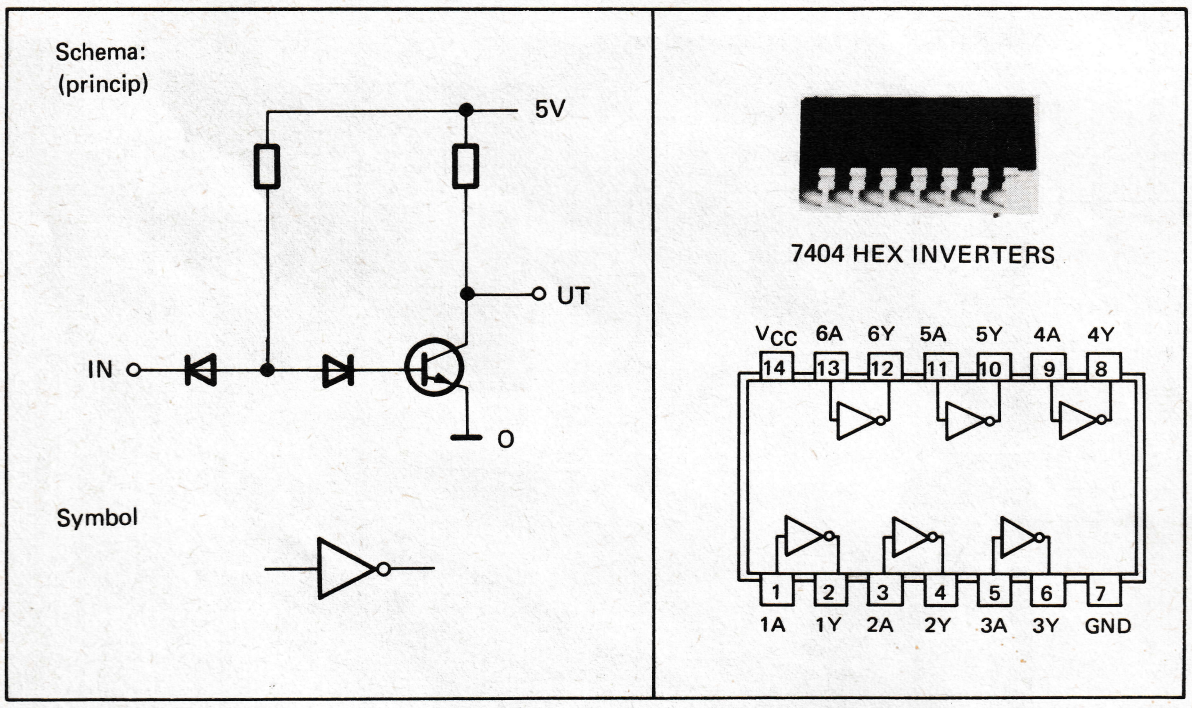


Fig 1.8 Inverterare.

Vi kan sammanfatta det ovan sagda på följande sätt:

- Vippor är kretsar som kan användas för lagring av siffror.
- Grindar är kretsar som kan användas för att öppna och stänga signalvägar, avkoda signalkombinationer eller utföra logiska operationer.

Vippor och grindar köps i dag färdigförpackade som IC-kretsar (IC = integrated circuit). Fig 1.9 visar några exempel på vippor och grindar ur de två för närvarande vanligaste logikfamiljerna, TTL och CMOS.

TTL betyder transistor transistor logik och en IC-krets av TTL-typ innehåller kretsar liknande dem vi visat i fig 1.4.

CMOS betyder komplementär MOS, där MOS står för metall oxid halvledare (semiconductor). CMOS-kretsar innehåller s k fälteffekttransistorer medan TTL-kretsar innehåller bipolära transistorer (dvs vanliga "gammeldags" transistorer).

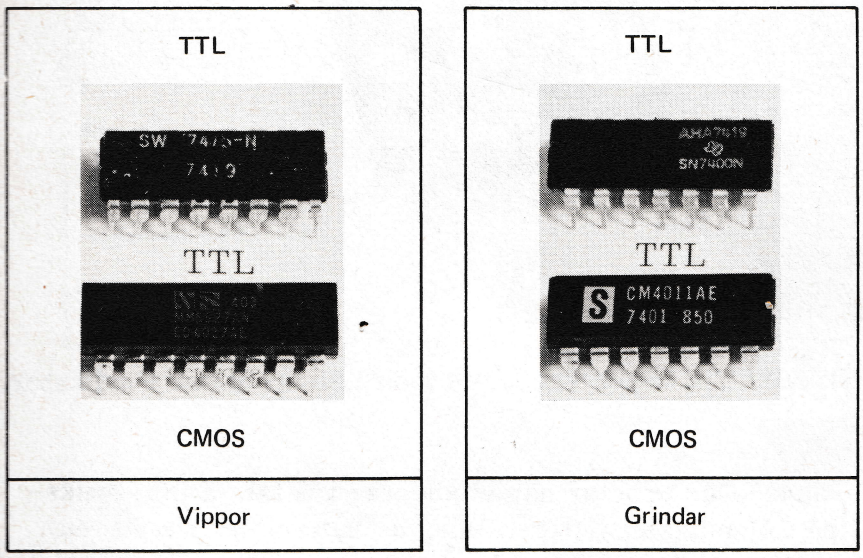


Fig 1.9 Vippor och grindar ur TTL- och CMOS-familjerna.

LSI-kretsar innehåller mängder av vippor och grindar som är hopkopplade (integrerade) till ett fungerande system. Kretsarna i en räknedosa är exempel på denna typ av IC-kretsar.

Nu är det dags att se på de funktioner som vippor och grindar fullgör i en kalkylator.

1.2 Räknedosan 10R

Fig 1.10 visar en vanlig räknedosa för de fyra räknesätten (Rockwell, typ 10R)

Tangentbordet innehåller två grupper av tangenter, siffertangenter (0-9) samt decimalkomma) och instruktionstangenter (C, +, -, x, ÷).

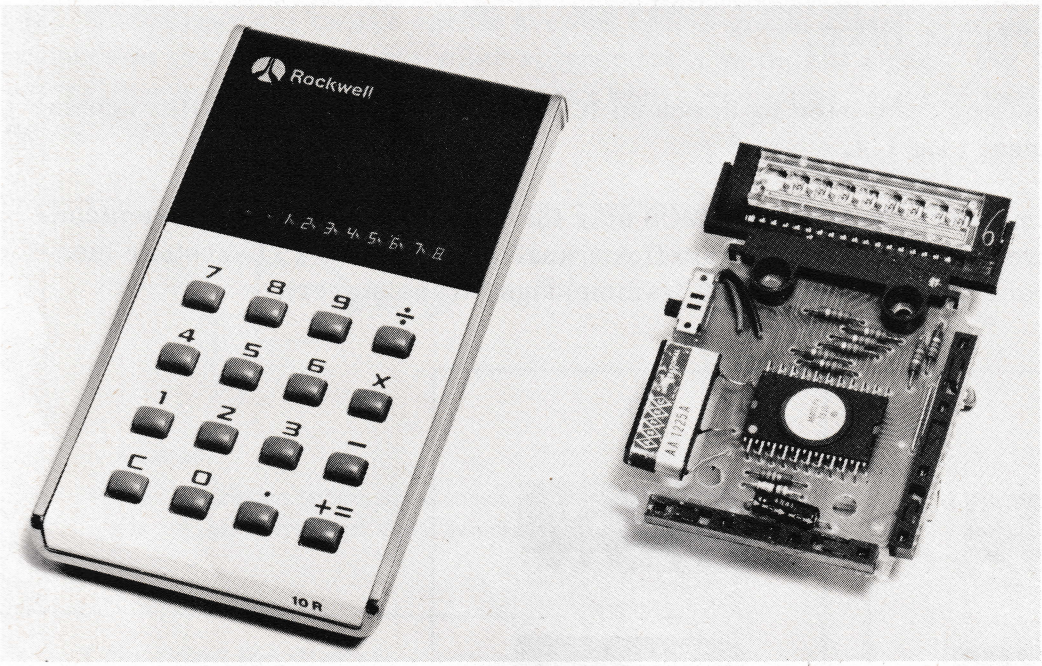


Fig 1.10 Räknedosan 10R (Rockwell). Till höger kretskort med LSI-krets och i indikator.

Vi ska nu bekanta oss med hur räknedosan arbetar när vi adderar två tal. Enligt bruksanvisningen går en addition till på följande sätt (efter tillslag av batteriströmbrytaren).

1. Nollställ räknaren med två tryck på nollställningstangenten C (clear).
2. Slå in det första talet (som på finare språk kallas augenden) på siffertangenterna.
3. Tryck på additionstangenten (+ =).
4. Slå in det andra talet (som kallas addend).
5. Tryck på resultatknappen (+ =) som i detta fall är samma knapp som additionstangenten.

Vad har nu ovanstående moment 1 - 5 för motsvarighet i de elektroniska kretsarna? För att besvara den frågan måste vi först sätta oss in i räknedosans blockschema, fig 1.11.

1.2.1 RÄKNEDOSANS BLOCKSCHEMA

Fig 1.11 visar blockschemat för räknedosan i fig 1.10. Schemat innehåller 6 block:

Indikator
Register A
Aritmetik och kontroll
Register B
Siffertangeter
Instruktionstangenter

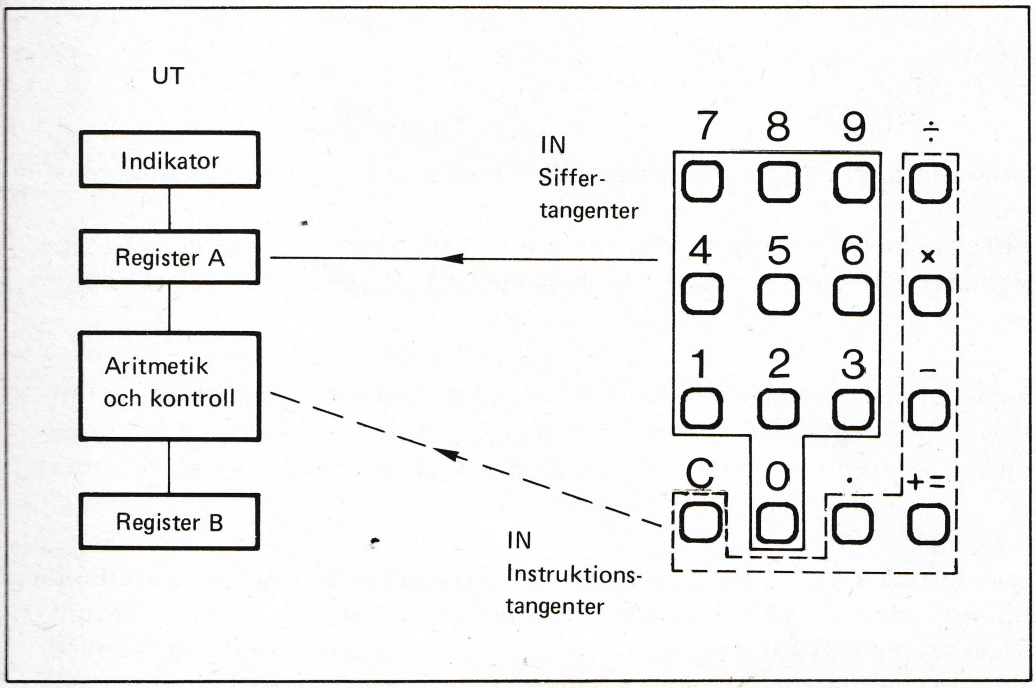


Fig 1.11 Räknedosans blockschema.

Via siffertangenterna matas register A med data (heldragen pil). Via instruktionstangenterna matas aritmetik- och kontrollenheten med instruktioner (streckad pil). Låt oss börja med att beskriva hur registren fungerar.

1.2.1.1 Registrens funktion

För att kunna ta emot två tal måste räknedosan ha två register, dvs minnen som kan lagra talen. I fig 1.11 kallas dessa för register A och register B. Vår räknedosa kan räkna med 8 decimala siffror. För att lagra en decimal siffra åtgår fyra binära vippor. Man använder därvid den BCD-kod (8421-kod) som framgår av fig 1.12. Varje register i fig 1.11 måste tydligen innehålla 8 · 4 = 32 vippor. Man säger att ett sådant register har storleken 32 bitar och att talen som lagras i registret har "ordlängden" 32 bitar.

Vikt 8 4 2 1	decimalt värde:
0 0 0 0	0
0 0 0 1	1
0 0 1 0	2
0 0 1 1	3
0 1 0 0	4
0 1 0 1	5
0 1 1 0	6
0 1 1 1	7
1 0 0 0	8
1 0 0 1	9

Fig 1.12 Den vanliga BCD-koden

Talen placeras i registren med hjälp av signalgrindar som styrs av kontrollenheten.

En räknedosas snabbhet behöver inte vara större än att vi får fram resultatet inom bekväm tid. En dator räknar däremot normalt i en helt annan takt (10^6 enkla operationer per sekund).

Eftersom kravet på snabbhet ej är särskilt stort är signalgrindarna i räknedosan så inkopplade att talen skjuts in i registret, siffra för siffra, från den minst signifikanta positionen (entalsiffran). Register som med hjälp av grindar på detta sätt kan skifta in tal kallas skiftregister. [x]

Innehållet i register A avkodas av ett antal grindar och indikeras med hjälp av lysdioder. Dessa lysdioder är placerade i ett sjusegmentsmönster (seven segment display) enligt fig 1.13. När samtliga segment lyser ser man en åtta. Övriga siffror kan tecknas med olika kombinationer av lysande segment. Omkodningen från vippornas BCD-kod till indikatorns sjusegmentsdisplay sker i logikkretsar av grindtyp.

---

x) En utförligare genomgång av ett skiftregisters funktion återfinns i Markesjö, Inledande kurs B, Läromedelsförlagen. Stockholm 1970, sid 37-38.

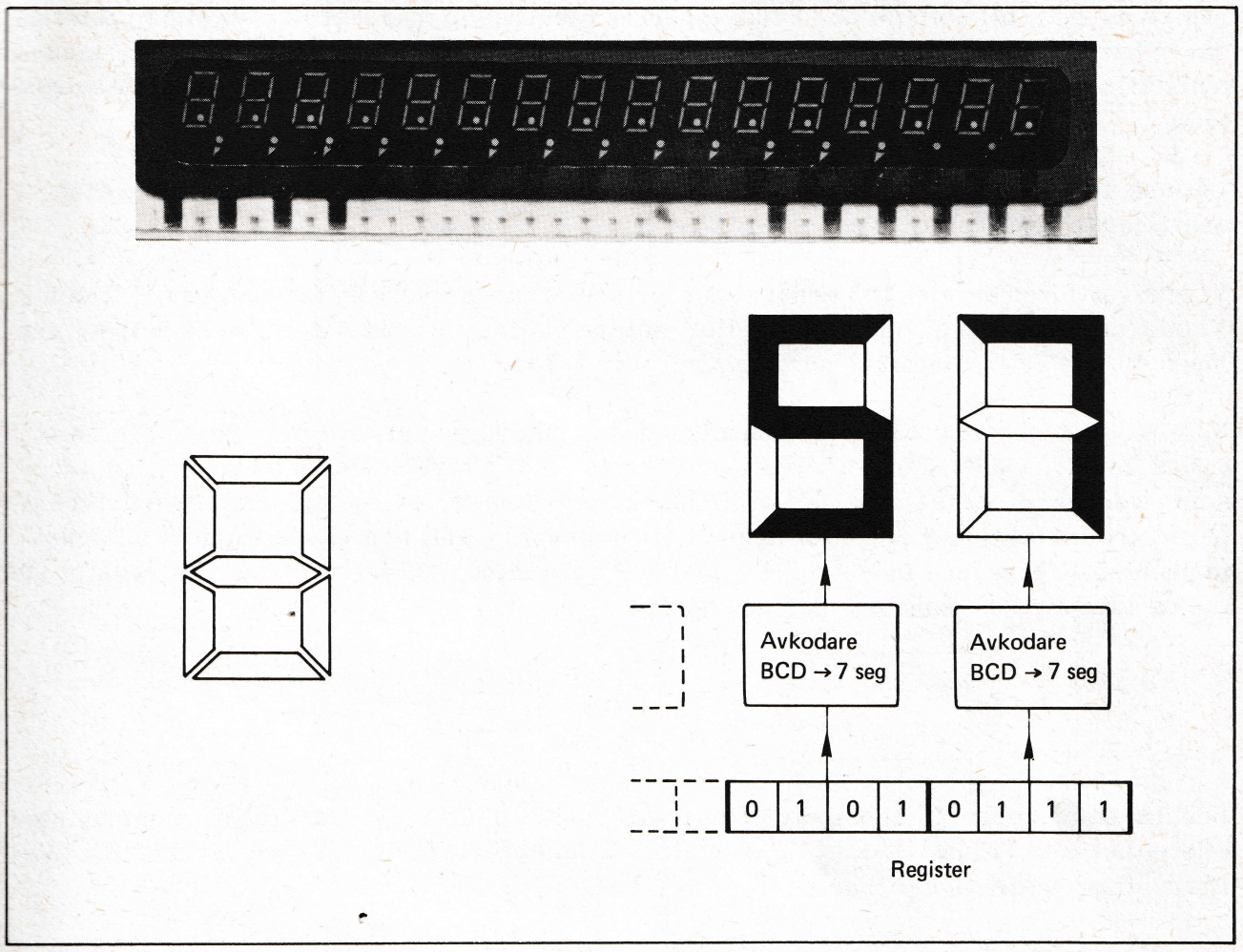


Fig 1.13 Indikator (display) av sjusegmenttyp.

1.2.1.2 Blockschematsuppbyggnad

Registrens uppgift är alltså att lagra tal. Men för att addera tal eller utföra andra aritmetiska operationer behövs en aritmetisk enhet. I fig 1.11 har denna enhet kombinerats med kontroll- eller styrenheten.

Blockschemat består alltså av två in-enheter (siffertangenterna och instruktionstangenterna), två register för lagring av tal, en räknare och kontrollenhet samt som ut-enhet ett display (eller indikator) som ständigt visar innehållet i register A.

Vi har nu fått en första inblick i räknedosans "elektroniska arkitektur" och nu ska vi göra en enkel addition. Vi ska addera talen 123 och 456.

1.2.2 ADDITION AV TVÅ TAL

Enligt räknedosans bruksanvisning skall vi börja med att nollställa räknaren med två tryck på C (clear)-tangenten. Det krävs två tryck om vi skall vara helt säkra på att båda

registren skall bli nollställda. Olika typer av räknedosor fungerar därvid olika. Somliga dosor har två nollställningstangenter, en C-tangent och en CE-tangent. Med C-tangenten nollställs därvid hela räknaren och med CE-tangenten (CE = clear entry) enbart A-registret.

Låt oss alltså anta att räknaren är nollställd och att vi kan börja våra egentliga räkneoperationer. Fig 1.14.

Vi skriver först in talet 123 genom att i tur och ordning trycka in tangenterna 1, 2 och 3. Vi ser hur talet 123 därvid skjuts (eller rättare skiftas) in från höger i A-registret vars innehåll hela tiden indikeras av displayet. Fig 1.14a.

Talet 123 är inskrivet och vi ger nu räknedosan instruktion att utföra addition genom att trycka in additionstangenten. Additionstangenten står i förbindelse med kontrollenheten som nu träder i verksamhet. Kontrollenheten utför en serie arbetsmoment (som ofta kallas mikroinstruktioner och som ligger i ett mikroprogram) och vi ska följa de viktigaste momenten. Först leds innehållet i register A (dvs talet 123) och register B (dvs talet 000) till den aritmetiska enheten, där de adderas:

```
 123
+000
 123
```

Adderaren är uppbyggd enbart av grindar och adderar här en siffra i taget. Summan som i detta fall blir 123, skiftas in i A-registret (från höger) samtidigt som det tidigare talet 123 skiftas ut (till vänster).

Via siffertangenterna tillförde vi alltså räknedosan data (talet 123) och med operationstangenten [±=] gav vi dosan instruktion att addera talen i A-registret och B-registret samt därefter sätta summan i A-registret.

Vi fortsätter nu med att skriva in talet 456 i A-registret, fig 1.14c. Så fort vi trycker ned en siffertangent efter en tidigare utförd addition flyttas hela innehållet i A-registret över till B-registret. (När vi i fig 1.14a skrev in talet 123 flyttades nollorna från A-registret över till B-registret. Eftersom det stod nollor i båda registren innebar detta ingen synbar ändring i registrens innehåll i fig 1.14a).

Efter inskrivning av talet 456 i A-registret har vi nu två tal i räknaren. 456 i A-registret och 123 i B-registret. Vid nedtryckning av additionstangenten uppenbarar sig denna gång ett nytt tal i indikatorn och det är lätt att se hur detta går till i fig 1.14d.

456	A-register
+123	B-register
579	A-register och indikator

Vi har därmed följt operationerna vid en enkel addition:

123	+	456	=	579
augend		addend		summa

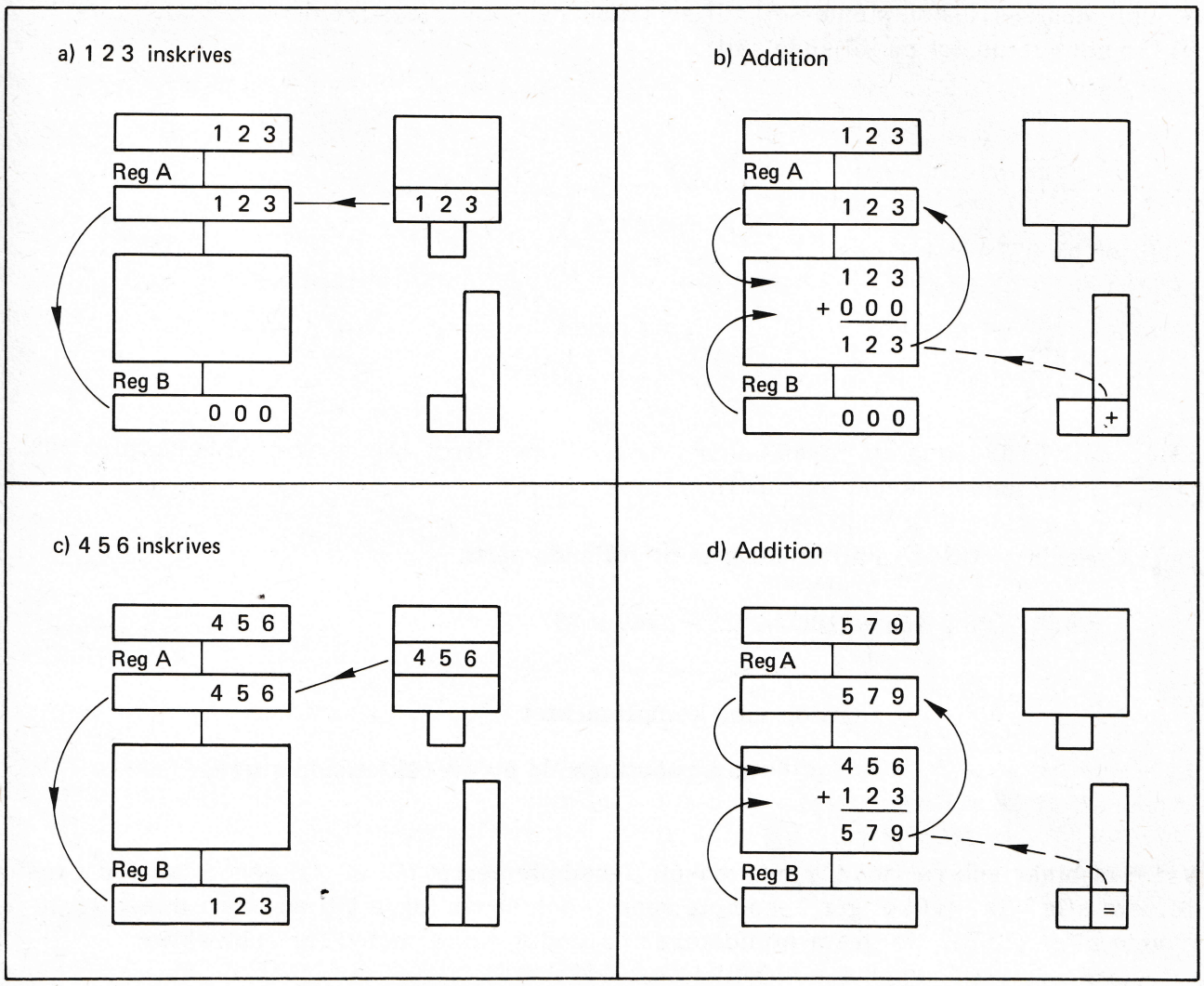


Fig 1.14 Addition av talen 123 och 456.
a) Inskrivning av talet 123
b) Addition
c) Inskrivning av talet 456
d) Addition.

1.2.3 DE FYRA RÄKNESÄTTEN

Vi har ovan enbart behandlat ett av de fyra räknesätt som räknedosan kan hantera. Räknedosan kan även subtrahera. Genom att ta komplementet av subtrahenden kan subtraktionen överföras till en addition och därmed kan samma kretsar som tidigare utförde additionen även användas för subtraktion. Låt oss se principen genom att utföra subtraktionen

456	-	123	=	333
minuend		subtrahend		differens

Låt oss för enkelhets skull i det fortsatta resonemanget anta att vår dosa enbart räknar med tre siffror.

Komplementet (10-komplementet) till subtrahenden 123 är vid tre decimala siffror 877. Vi får komplementet på följande sätt:

$$\begin{array}{r} 1000 \\ \underline{-123} \\ 877 \end{array}$$

Addition av 877 till 456 ger

$$\begin{array}{r} 456 \\ \underline{+877} \\ 1333 \end{array}$$

Eftersom vi här antog att dosan enbart tar med tre siffror blir svaret på indikatorn 333. (Pröva själv med ett annat exempel).

Decimal subtraktion kan alltså utföras på följande sätt:

$$456 - 123 = 456 + \underbrace{(1000-123)} - 1000 = 333$$

addition med komplementet

samt borttagande av överskjutande siffra

Vid rent binär subtraktion får man enkelt 2-komplementet till ett tal genom att först invertera alla bitar (vilket ger 1-komplementet) och sedan lägga till en etta i minst signifikanta biten (LSB). De flesta minidatorer använder denna metod för subtraktion.

Multiplikation erhålles i vår räknedosa genom upprepad addition och skift. Division utföres genom upprepad subtraktion och skift.

Räknedosan innehåller alltså fyra mikroprogram, ett för vardera av de fyra räknesätten. Varje sådant mikroprogram startas av respektive instruktionstangent. Dessutom finns en tangent för nollställning av samtliga register.

1.3 Några viktiga begrepp

Räknedosan i vårt exempel illustrerar på ett enkelt sätt innebörden hos några viktiga begrepp:

Begrepp:	I vårt exempel:
inenhet (input unit):	10 siffertangenter och 5 instruktionstangenter
utenhet (output unit):	indikator
register (register):	A- och B-register som lagrar tal
aritmetisk enhet: (kallas ofta ALU, aritmetic logic unit)	adderare
kontrollenhet (control unit):	styrenhet som öppnar de vägar som talen i registren skall passera vid adderingen
data (data):	tal som skrivs in i A-registret eller kommer ut i indikatorn
instruktion (instruction):	addera
mikroinstruktion (microinstruction):	delar av additionsinstruktion, exempelvis skifta talen i A- och B-registren till adderaren, addera, skifta summan till A-registret
mikroprogram (microprogram):	utför i rätt ordningsföljd de mikroinstruktioner som behövs för exempelvis en addition
arkitektur (architechure):	anger i en skiss funktionsblocken (register, adderare m fl) samt de viktigaste data- och instruktionsvägarna
kod (code):	binär kod, BCD-kod
avkodare (decoder)	för omvandling av binära till decimala tal och för indikering i 7-segment display
ordlängd (word length):	32 bitar eller 8 decimala siffror i A-registret

Det är givetvis många detaljer vi hoppat över när det gäller räknedosans funktion. Meningen har här endast varit att peka på några viktiga begrepp och funktionsenheter som förekommer både i samband med räknedosor och minidatorer.

Vår räknedosa i fig 1, 10 har fördelen av att ha fast inbyggda program och den är därmed mycket enkel att använda. Den är däremot starkt begränsad i två olika avseenden.

- Utöver nollställning kan vår räknedosa enbart utföra fyra rutiner (program) - de fyra räknesätten.
- Vår räknedosa saknar minne. Vi kan inte lagra upp och spara olika delresultat under beräkningens gång.

För att täcka behovet av fler instruktioner och större minne har det kommit fram en rad räknedosor på marknaden som ofta kallas "scientific calculators". Fig 1.15 visar som exempel en dosa som utöver de fyra räknesätten har instruktionstangenter för de funktioner som finns på vanliga räknestickor samt dessutom några statistiska funktioner. Vi ska emellertid nu hoppa ett steg längre upp på skalan av olika kalkylatorer och bekanta oss med "världens minsta dator", en programmerbar kalkylator med många användningsområden.

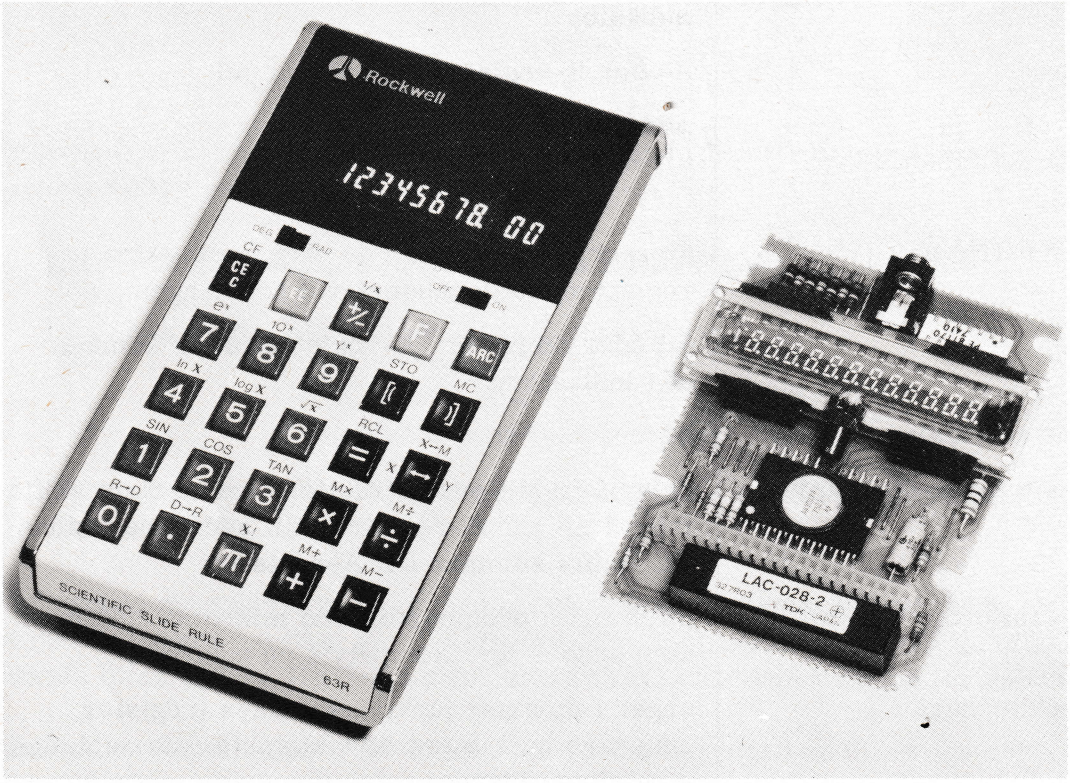


Fig 1.15 Räknedosa med samma funktioner som på en vanlig räknesticka (kretskort med LSI-krets och indikator till höger på bilden).

2. En programmerbar kalkylator

Fig 2.1 visar den programmerbara kalkylatorn Compucorp 326. Skillnaden mellan Compucorp 326 och den enkla räknedosan 10R är inte så stor rent principiellt men skillnaden är däremot avsevärd när det gäller den praktiska användningen.

Fig 2.1 Kalkylator med kassettbandminne.

2.1 Kalkylatorns blockschema

Låt oss först titta på blockshemat när kalkylatorn används som en enkel räknare, fig 2. 2. Vi lägger märke till två skillnader jämfört med den enkla räknedosan.

- Kalkylatorn innehåller ett minne. Vi kan med dess hjälp lagra 12 stycken tal (vardera med 13 siffror)
- Kalkylatorn innehåller ett stort antal instruktionstangenter med olika funktioner.

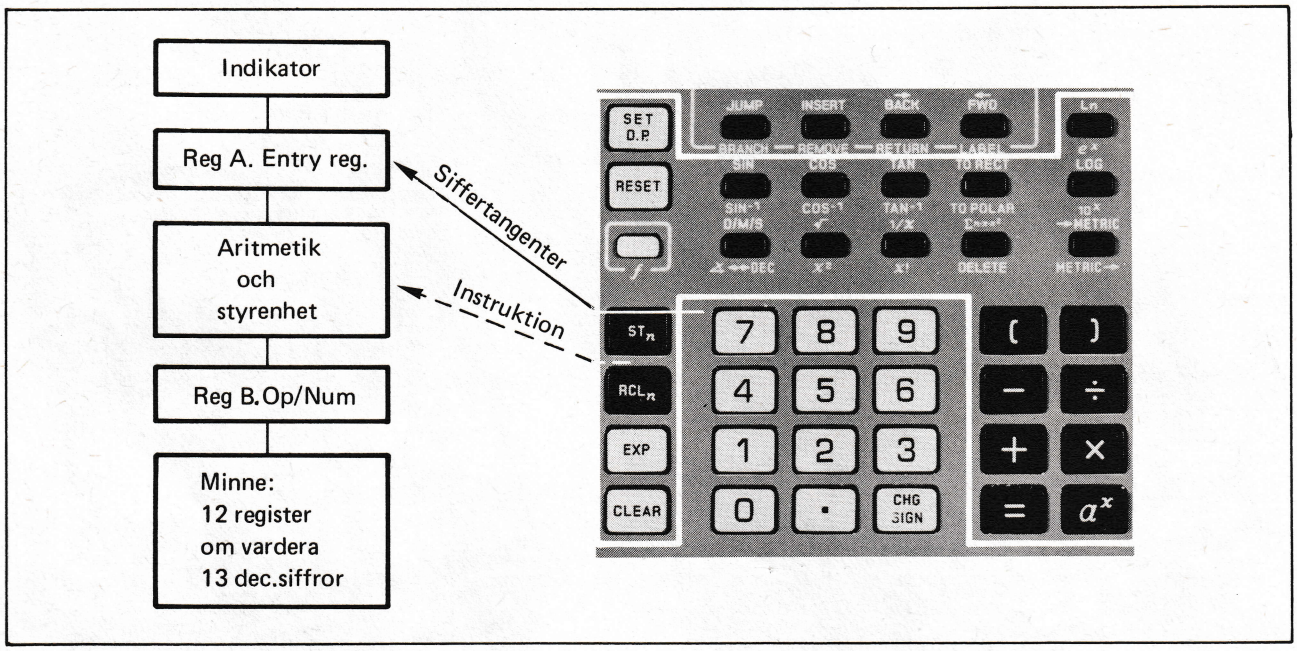


Fig 2. 2 Blockschemat för den programmerbara kalkylatorn Compucorp 326 (programminnet utelämnat).

Vill vi lagra det tal som för tillfället står inskrivet i A-registret behöver vi enbart göra två nedtryckningar:

- Ge kalkylatorn instruktion om lagring med tangenten [ST$_n$] (dvs "store" innehållet i A-registret i register n)
- Ge kalkylatorn adressen till det register (n) i vilket vi vill ha talet lagrat. Detta sker genom nedtryckning av en av de 12 siffertangenterna ([.] och [CHG SIGN] räknas här som siffertangenter).

Vill vi vid ett senare tillfälle använda det tal vi lagrat i register n behöver vi enbart trycka på lästangenten [RCL$_n$] (dvs "recall" n) och därefter på den siffertangent som anger adressen.

Antag till exempel att vi tidigare lagrat talet 123 i register 7. Vi får då tillbaks talet till A-registret med följande två tangentoperationer:

[RCL$_n$] (kopiera talet i minnesregister n till A-registret)

[7] (n = 7)

Vi ser nu det tidigare lagrade talet 123 i indikatorn

2.2 Addition av två tal

En addition av två tal kan göras exakt på samma sätt i kalkylatorn 326 som vi tidigare beskrivit för vår enkla räknedosa 10R. Vi ska emellertid nu använda minnet vid additionen. Vi tänker oss att talet 123 i samband med tidigare beräkningar har blivit inskrivet i minnesregister nr 1 (med adressen n = 1) och talet 456 i minnesregister 2. I samband med en ny beräkning vill vi dels räkna ut summan av dessa två tal och dels lagra summan för att ha den tillgänglig vid ett senare tillfälle. Låt oss följa denna enkla räknerutin.

För att få både A- och B-registren nollställda trycker vi först in RESET-tangenten. (CLEAR-tangenten nollställer enbart A-registret). Allt utom de 12 minnesregistren är nu nollställt.

Först läser vi in talet från minnesregister 1 till räkneregister A:

[RCL$_n$] (kopiera till A)

[1] (n = 1)

och i displayet ser vi nu talet 123 som kopierats från minnesregister 1 till A-registret. (123 står fortfarande kvar i minnesregister 1).

Nu ger vi additionsinstruktion genom att trycka in additionstangenten

[+] (addera)

Därvid kopieras A-registrets innehåll till B-registret och samtidigt kallas det mikroprogram in som skall utföra additionen.

Vi skall addera innehållet i minnesregister 2 och kopierar därför över detta till A-registret:

[RCL$_n$] (kopiera till A)

[2] (n = 2)

Nu innehåller alltså A-registret tal 456 (från minnesregister 2) och B-registret talet 123 (från minnesregister 1).

Vid nedtryckning av resultatstangenten

[=] (beräkna resultatet)

sätts det tidigare inkallade mikroprogrammet för addition i verksamhet. Och vi får summan 579 i A-registret.

Det är nu enkelt att lagra summan i minnesregister 3:

[ST$_n$] (lagra A)

[3] (n = 3)

Även om vi nu nollställer kalkylatorn med RESET-tangenten och utför en rad andra beräkningsarbeten så har vi resultatet sparat i minnesregister 3.

2.3 Programmering av en enkel addition

Kalkylatorn 326 är programmerbar. Den har ett speciellt programminne som omfattar 160 steg och som vi ritat in i fig 2. 3.

Om vi ofta i vårt räknearbete behöver göra en viss rutin, exempelvis summera talen i minnesregister 1 och minnesregister 2, kan vi skriva in ett program i programminnet som utför denna räknesekvens automatiskt då vi ger startkommando, dvs då vi trycker på [START/STOP] -knappen.

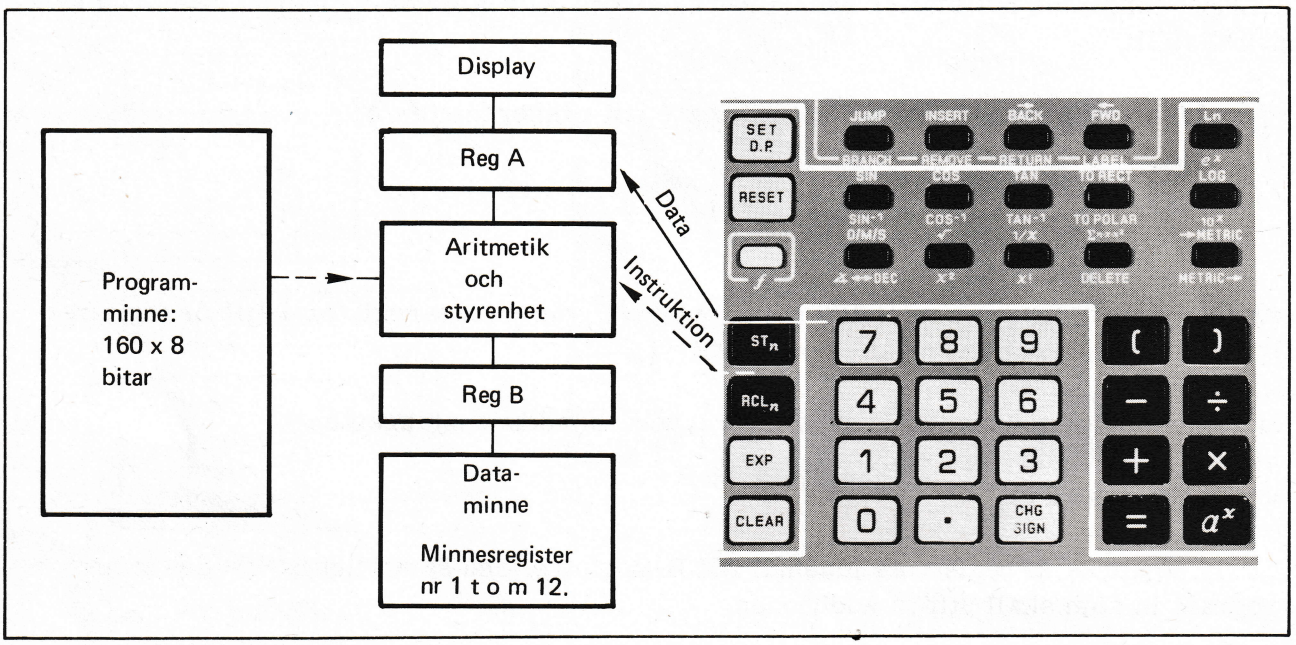


Fig 2. 3 Blockschema för den programmerbara kalkylatorn 326.

Låt oss först se rent praktiskt hur detta tillgår.

För att få access (dvs bli inkopplad) till programminnet ställer vi tangenten

[—] run
step
load

(run = starta
step = stega
load = ladda)

i läge LOAD.

Varje tangentnedtryckning vi gör kommer nu att registreras i programminnet.

Den rutin vi vill programmera in är alltså följande:

[RESET] Nollställ kalkylatorn

[RCL$_n$] [1] } Kopiera talet i minnesregister 1 (dvs adress n = 1) till A-registret

[+] Addera (med följande)

RCL$_n$ 2 — Kopiera talet i minnesregister 2 (dvs adress n = 2) till A-registret

= — Beräkna resultatet

ST$_n$ 3 — Lagra resultatet i minnesregister 3 (dvs adress n = 3)

START/STOP — Stopp

Det enda moment vi inte känner igen från det tidigare är stop-ordern. Men det är ju nu fråga om att köra en rutin och vi måste därför tala om för kalkylatorn när den har fullgjort den önskade uppgiften och kan stanna.

Efter nedtryckningen av dessa tangenter slår vi över omkopplaren i läge RUN

run
step
load

Kalkylatorn är nu programmerad att utföra vår enkla additionsrutin. Låt oss pröva!

Först sätter vi in talen 123 och 456 i register 1 respektive 2.

Kommentar	Tryck på tangent	Indikatorn visar
nollställ	RESET	0.0000
talet 123	1	1.0000
	2	12.0000
	3	123.0000
lagras i minnet på adress n = 1	ST$_n$	123.0000
	1	123.0000
talet 456	4	4.0000
	5	45.0000
	6	456.0000
lagras på n = 2	ST$_n$	456.0000
	2	456.0000
nollställ	RESET	0.0000

Kalkylatorn är nu nollställd men i dess minne finns talen 123 och 456 inlagrade i minnet på adresserna n = 1 resp n = 2.

Trycker vi nu på START/STOP -knappen verkställs vårt tidigare inskrivna program.

Indikatorn visar nu direkt summan 579 och om vi nollställer kalkylatorn med RESET-knappen så finns fortfarande denna summa kvar i minnesregister 3. Låt oss pröva!

Tryck på tangent	I displayet står
RESET	0.0000
RCL$_n$	0.0000
3	579.0000

Den rutin vi programmerade in i kalkylatorns programminne ser vi i vänstra kolumnen i fig 2.4.

Instruktion	Programräknare	Instruktionskod	Kommentar
RESET	1	036	Nollställ register A och B
RCL$_n$	2	310	Läs minnesregister nr
1	3	001	1 till register A
+	4	021	Addera
RCL$_n$	5	310	Läs minnesregister nr
2	6	002	2 till register A
=	7	020	Beräkna resultatet
ST$_n$	8	300	Lagra register A i
3	9	003	minnesregister nr 3
START STOP	10	033	Stopp

Fig 2.4 Ett enkelt additionsprogram

Programminnet rymmer som nämnts totalt 160 programsteg (eller instruktioner). Vi har endast utnyttjat de 10 första positionerna i programminnet för vårt program.

För att kalkylatorn skall kunna hålla ordning på instruktionerna har den en programräknare (PC, program counter). Den räknas fram ett steg för varje instruktion som exekverats. Vi ser programräknarens ställning i andra kolumnen i fig 2.4.

Instruktionerna lagras i programminnet i en tresiffrig oktal kod. Det decimala talet 9_{10} blir om det skrivs i oktal kod 11_8 (indexen anger basen i resp talsystem) dvs 8-talsiffran 1 och entalsiffran 1. Vilket decimalt tal avses med beteckningen 27_8?
($2 \times 8 + 7 \times 1 = 23_{10}$).

Indikatorn visar i läge LOAD programräknarens ställning (decimalt) med tillhörande instruktion samt föregående och efterföljande instruktioner. Vi har på detta sätt möjlighet att kontrollera programmet. I fig 2.5 visas när programräknaren är i läge 007 (talet

som föregås av en punkt). Omedelbart till höger om programräknaren ser vi den instruktionskod som är lagrad på plats 7 i programminnet. Föregående och efterföljande instruktioner syns längst till vänster resp längst till höger i fig 2.5 (jämför med fig 2.4!).

Fig 2.5 Programsteg 6, 7 och 8 från fig 2.4 synliga i displayet.

Vår kalkylator är enkel att programmera och den har kraftfulla instruktioner. Vi har i vårt enkla exempel enbart utnyttjat en bråkdel av dess möjligheter. I fig 2.6 återges instruktionslistan som totalt omfattar 183 instruktioner (inklusive siffror som används som konstanter i programmet).

Utöver aritmetiska operationer (exempelvis addition som vi använt i vårt exempel ovan) kan kalkylatorn utföra hopp (jump) i programmet. Den kan exempelvis hoppa till önskat programsteg om innehållet i A-registret uppfyller vissa villkor. Kalkylatorn kan också programmeras att utföra subrutiner (branch - return) och hoppar därvid ifrån (branch) sin arbetssekvens i huvudprogrammet till en subrutin (som kanske används ett flertal gånger under beräkningen) och återgår (return) till huvudprogrammet när subrutinen har utförts.

000	0	023	×	071	COSINE	173	TO POLAR	304	ST ÷
001	1	024	÷	072	TANGENT	200 ↓ 213	f LABEL; 0 TO CHG SIGN	305	ST a^x
002	2	025	a^x	073	TO RECT.	220 ↓ 234	SET D. P.; 0 TO EXP	310	RCL
003	3	026	(	100 ↓ 114	f; 0 TO EXP	240 ↓ 253	→ METRIC; 0 TO CHG SIGN	311	RCL +
004	4	027	)	115	f, D/M/S	260 ↓ 273	f; METRIC →; 0 TO CHG SIGN	312	RCL −
005	5	030	RETURN	137	CLEAR ALL REGS	300	ST	313	RCL ×
006	6	033	START/STOP	140	PREBLOCK	301	ST +	314	RCL ÷
007	7	036	RESET	150	Σ DELETE	302	ST −	315	RCL a^x
010	8	037	CLEAR	160	e^x	303	ST ×	320	EXCH
011	9	040	WRITE PROG.	161	10^x			321	EXCH +
012	•	041	READ TAPE	162	x^2			322	EXCH −
013	CHG SIGN	050	$\Sigma n \cdot x \cdot x^2$	163	$x!$			323	EXCH ×
014	EXP	060	Ln	170	ARC SINE			324	EXCH ÷
015	D/M/S	061	LOG	171	ARC COSINE			325	EXCH a^x
020	=	062	$\sqrt{}$	172	ARC TANGENT			340	WRITE DATA
021	+	063	$1/x$					350	JUMP
022	−	070	SINE					351	JUMP +
								352	JUMP −
								353	JUMP + −
								354	JUMP =
								355	JUMP + =
								356	JUMP − =
								357	JUMP + − =
								360	BRANCH
								361	BRANCH +
								362	BRANCH −
								363	BRANCH + −
								364	BRANCH =
								365	BRANCH + =
								366	BRANCH − =
								367	BRANCH + − =

→ METRIC	AND f METRIC →	CODE TO METRIC	CODE FROM METRIC
0	°F→°C	240	260
1	INCH→CM	241	261
2	FOOT→METRE	242	262
3	MILE→KILOMETRE	243	263
4	IN³→CM³	244	264
5	US GAL→LITRE	245	265
6	UK GAL→LITRE	246	266
7	POUND→KILOGRAM	247	267
8	OUNCE→GRAM	250	270
9	LB/FT³→GM/CM³	251	271
•	LB/IN²→KG/CM²	252	272
CHG SIGN	DEG (GRAD)→RAD	253	273

f	FUNCTION	CODE
0	CLEAR REGS 1,2,3	100
1	DISPLAY e	101
2	ABSOLUTE VALUE	102
3	MEAN	103
4	STD. DEV.	104
5	FRACTION	105
6	INTEGER	106
7	DISPLAY π	107
8	ROUND DISPLAY	110
9	IDENTIFIER	111
CHG SIGN	AUTO TEST	113
EXP	PAUSE	114
CLEAR	CLEAR ALL REGS	137

326 SCIENTIST

Fig 2.6 Instruktionslistan till kalkylatorn 326.

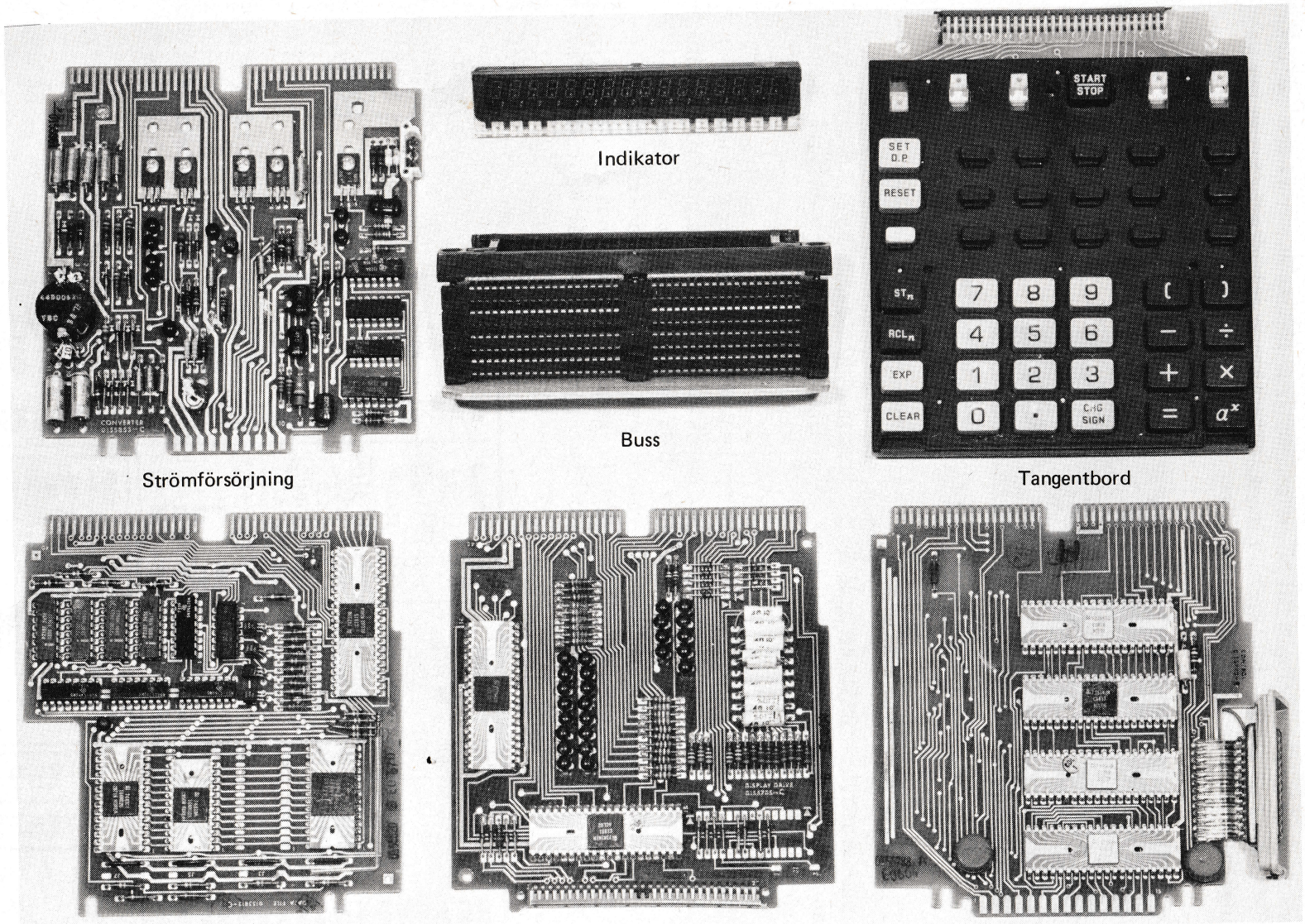


Fig 2.7 IC-kretsarna i Compucorp 326.

2.4 Kalkylatorns uppbyggnad

Compucorp 326 är uppbyggd av högintegrerade kretsar. Fig 2. 7 visar innandömet i kalkylatorn.

Kretsarna är monterade på fem kort varav ett innehåller CPU:n (dvs aritmetik och styrenhet i fig 2. 3) och ett innehåller 10 halvledarminnen (6 ROM och 4 RAM).

Kalkylatorn tar mycket liten effekt och kan därför matas från ett par inbyggda små ackumulatorer, fig 2. 8. På dessa ackumulatorer kan kalkylatorn köras ca 4 timmar utan omladdning.

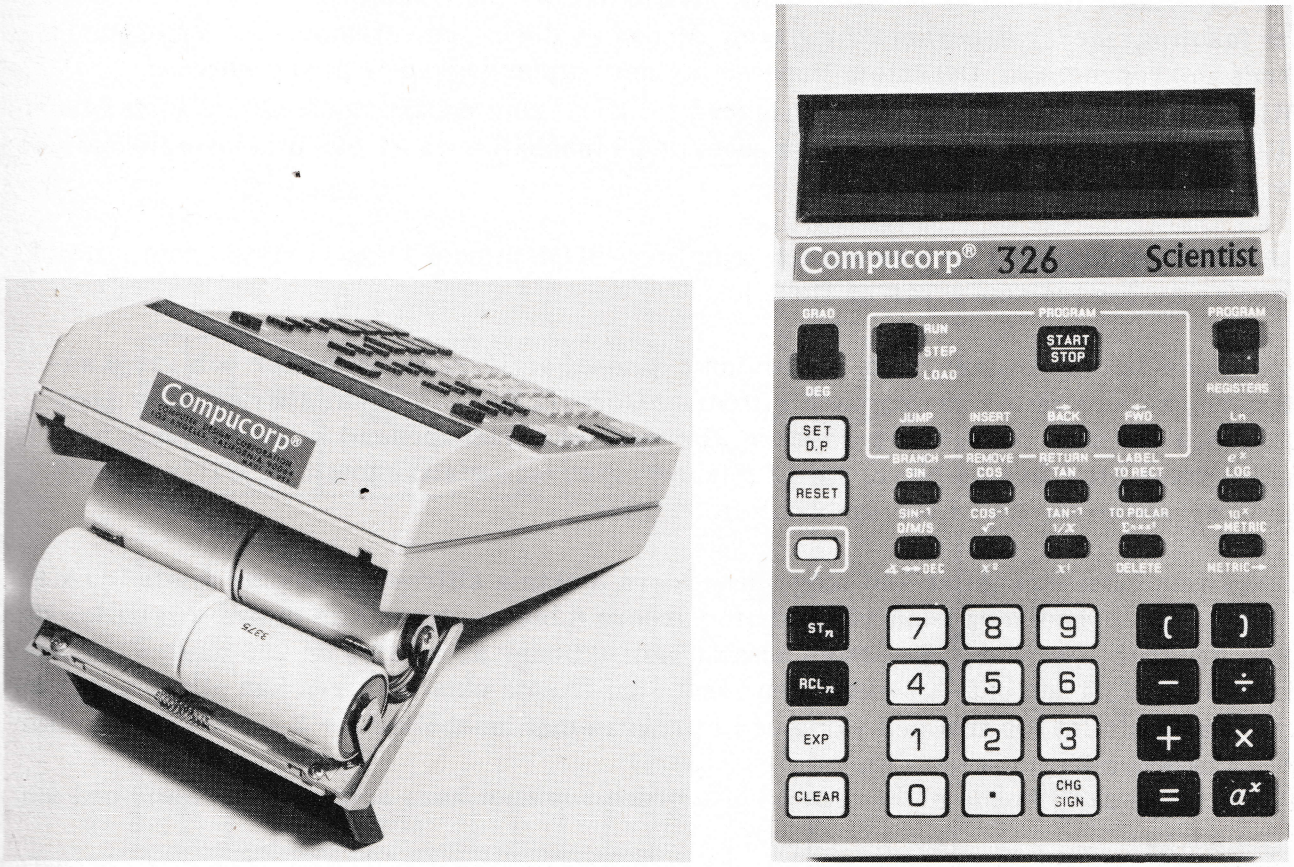


Fig 2. 8 Kalkylatorn matas från fyra små ackumulatorer.

2.5 Kalkylatorns minnen

Vi har tidigare omnämnt att vår kalkylator innehåller två minnen, ett dataminne omfattande 12 minnesregister och ett programminne omfattande 160 steg.

Typiskt för dessa minnen är att man själv både kan skriva in information till och läsa ut information från minnet. Man kallar denna typ av minnen för skriv-läsminnen.

RAM är en annan vanlig benämning på skriv-läsminnen. RAM betyder random access memory och med detta avses att man har lika snabb access till alla positionerna i minnet. Det finns totalt ca 4000 bitar RAM-minne i Compucorp 326.

Vår kalkylator innehåller ett stort antal instruktioner med komplexa funktioner (exempelvis fakultet, medelvärdesberäkning etc). Många av dessa instruktioner kräver ett omfattande mikroprogram. Dessutom finns en hel del rutiner lagrade i programform.
All denna information är permanent lagrad i minnen som enbart kan läsas, s k läsminnen eller ROM (read only memory). Compucorp 326 innehåller ca 40 000 bitar programminne av ROM-typ.

RAM-minnet inryms i fyra kristaller (chip) och ROM-minnet i sex. Kristallerna sitter i kapslar av den typ som visas på fig 2. 9.

Ett minne är organiserat i ord. Dataminnet innehåller 12 minnesregister och varje sådant register omfattar 13 decimala siffror. Eftersom varje decimal siffra kräver fyra bitar och varje tal dessutom två tecken (för mantissa och exponent) kommer varje minnesregister att kräva 64 bitar. De 12 minnesregistren omfattar totalt 12 x 16 x 4 = 768 bitar organiserade på 12 ord om vardera 64 bitar.

Programminnet innehåller 160 steg om vardera 3 oktala siffror och upptar totalt 160 x 8 = 1280 bitar organiserade på 160 ord om vardera 8 bitar. Dessa åtta bitars ord kallas ofta byte (oktader, oktetter). Eftersom antalet instruktioner är mindre än 256 räcker det med talen 0 - 377_8 i den oktala koden. Den mest signifikanta oktala siffran kan alltså kodas med två bitar (istället för tre) och varje ord i programmet innehåller 2 + 3 + 3 = 8 bitar (eller 2 1/2 oktala siffror). Fig 2. 10.
För att man skall kunna hålla reda på alla orden i minnet har, som vi sett, varje ord sin egen adress.
När instruktionerna i ett program successivt exekveras är det programräknarens uppgift att tillhandahålla adressen för varje nytt programsteg. Därför räknas programräknaren upp ett steg för varje ny instruktion.

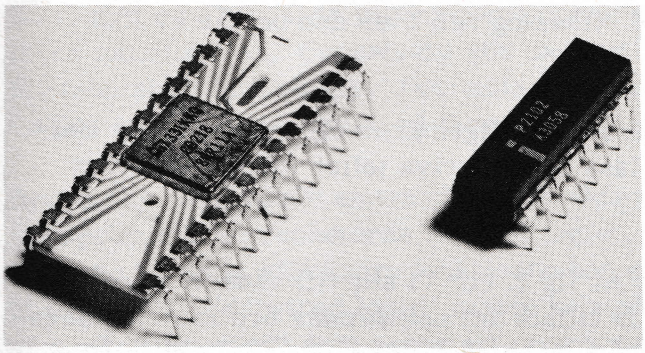
RAM-minnet

Ett elektroniskt minne av RAM-typ utgörs i princip av ett antal vippor.[x)] Dessa vippor är ordnade i ord och varje ord har en given adress. Karakteristiskt för en minneskrets är därför att den har två typer av ingångar, adressingång och dataingång. Adressingången består av ett antal ledningar (bitar) som är tillräckligt stort att koda samtliga ords adresser (för 12 ord åtgår minst 4 bitar eftersom $2^4 > 12 > 2^3$).

Dataingången används vid inläsning i minnet och har lika många ledningar som ordläng-

---

x) I s k dynamiska minnen utgörs varje bit i princip av en kondensator.



ROM RAM

Fig 2.9 Halvledarminnen av typen ROM och RAM.

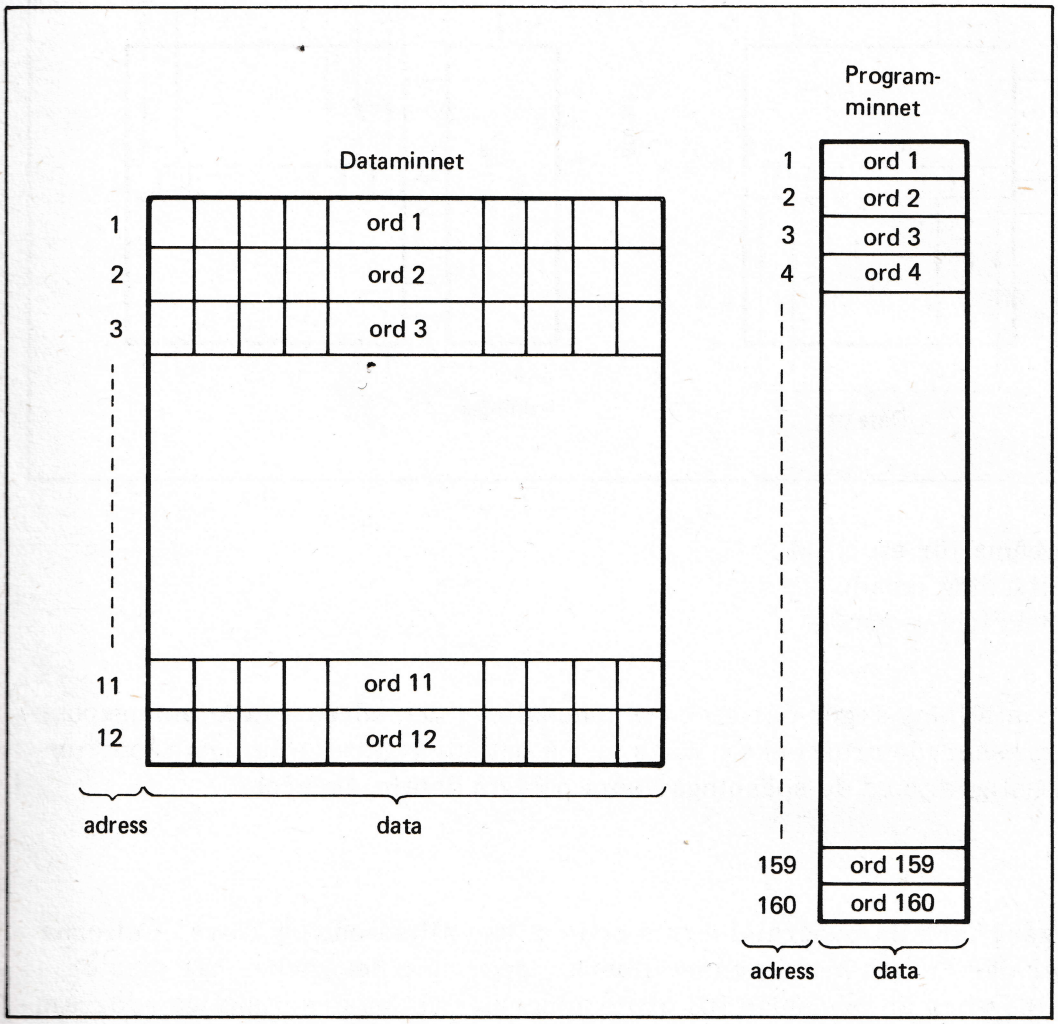


Fig 2.10 Minnets organisation.

den (räknat i bitar). Datautgången används vid utläsning och även den består av lika många ledningar som ordlängden i bitar.

Ofta finns en tredje typ av ingångar till ett minne, nämligen kontrollingångar. Via kontrollingångarna ges minnet order om det ska läsa eller skriva (eller om det ska vara helt frånkopplat).

Blockschemat för ett halvledarminne framgår av fig 2.11. Vi ser till vänster adressingångarna och adressavkodaren. De bitar som tillhör ett och samma ord har en gemensam ledning, ordledningen. Adressavkodaren fungerar så att den påverkar spänningen på enbart den ordledning vars adress är pålagd adressingången (i form av en kombination av höga och låga spänningar på adressledningarna).

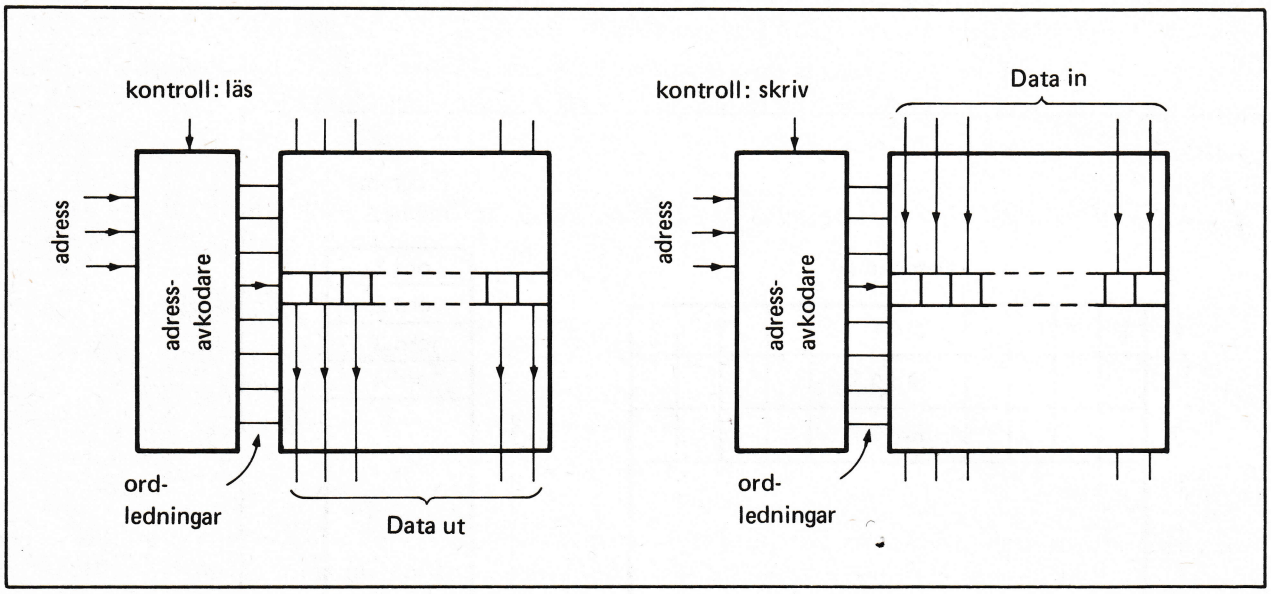


Fig 2.11 Blockschema för ett RAM
a) vid utläsning (read)
b) vid inskrivning (write)

Kontrollingångens ställning avgör därefter om innehållet i den adresserade minnespositionen (dvs det adresserade ordet) skall läsas ut via datautgångarna eller om vipporna skall ställas in i enlighet med de spänningar som pålagts dataingångarna.

ROM-minnet

Ett ROM har en gång för alla programmerats och det kan alltså enbart läsas. Cellerna i ett ROM behöver därför inte bestå av inställbara vippor utan det räcker här med en enkel transistor vars bas är inkopplad till ordledningen. Vid den ursprungliga programmeringen av ett ROM gör man ett avbrott i basledningarna till de transistorer som inte skall leda och på det viset får man önskat ord permanent inskrivet på den givna adressen.

Eftersom ROM kräver färre komponenter än RAM kan man i en kristall av given storlek få in betydligt fler bitar i ett ROM än i ett RAM.

En fördel med att lägga mikroprogram och rutiner i ROM är att fabrikanten lätt kan variera funktionerna. Compucorp gör exempelvis speciella kalkylatorer för statistiska och för tekniska beräkningar. Det enda som skiljer dessa kalkylatorer åt är programmen i ROM-minnena och texten på tangenterna.

Det är för närvarande en trend inom elektroniken att övergå från de tidigare av grindnät uppbyggda styrenheterna ("wired logic", "random logic") till programstyrda kretsar. Man kan på detta sätt ändra funktionen hos en elektronisk apparat enbart genom att byta ut ett ROM. Vid "wired logic" måste hela elektroniken byggas om. Man hör ibland begreppet "firmware" som avser att programmet är permanent lagrat i ROM. Firmware är alltså ett mellanting mellan "software" (där programmet läses in) och "hardware" (där programmet är inbyggt vid konstruktionen av kretsen och därför aldrig kan ändras). "wired logic" kallas ofta "fast logik" på svenska.

Halvledarminnen har de senaste åren blivit en av de viktigaste komponenterna inom elektroniken.

2.6 Bandminnet

Datorer har normalt yttre minnen i form av skiv-, trum- eller bandminnen för lagring både av program och data.

Till vår kalkylator 326 finns en liten kassettbandstation, Compucorp 392, som med en enkel kontakt kan pluggas in till kalkylatorn.

Samtliga inre minnesregister (och även programminnet i kalkylatorn) är halvledarminnen - dvs i princip uppbyggda av vippor av den typ vi tidigare studerat. När vi slår ifrån strömmen till kalkylatorn förlorar vi innehållet i dessa vippor. Man säger att halvledarminnen är "volatila", dvs tappar minnet när strömtillförseln upphör. I ett bandminne kan däremot innehållet förvaras permanent eftersom det är lagrat genom magnetisering av ett band.

Om vi för framtida bruk vill spara ett program som vi tidigare skrivit in i kalkylatorns programminne, kan vi spela över det från programminnet i kalkylatorn till kassettbandet i bandstationen.

Fig 2. 12 Kassettbandstation (Compucorp 392) till kalkylatorn.

Vi ska nu följa hur en sådan inspelning går till. Den är samtidigt ett exempel på hur en I/O-enhet fungerar (I/O= input/output) och man talar när det gäller datorer ofta om kringutrustning (peripherals) som inkopplas till datorn via "interface"-kretsar. I vårt fall är kringutrustningen en enkel bandspelare och anpassningskretsarna (interface circuits) är inbyggda i kalkylatorn.

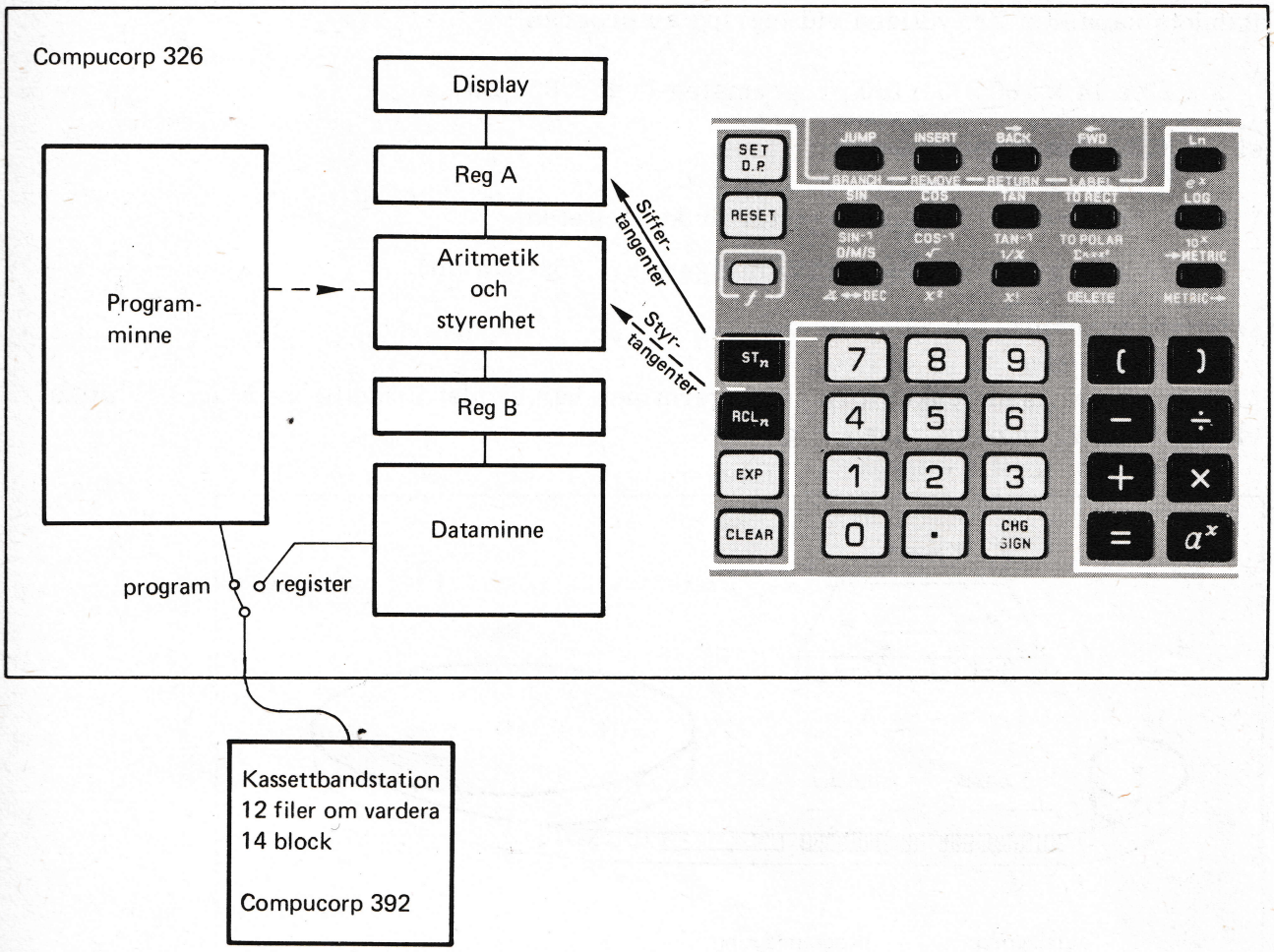


Fig 2.13 Blockschemat för kalkylator med inkopplad kassettbandstation.

2.6.1 FORMATERING

Innan man kan spela in data eller program på vår kassett måste bandet "märkas upp" eller formateras i numrerade fack (preconditioning). I kalkylatorn finns en rutin för detta ändamål lagrad i ett av ROM:en. Med hjälp av denna rutin kan pulser spelas in på bandet på ett sådant sätt att bandet blir uppmärkt i 12 filer och varje sådan fil blir indelad i 14 block.

Vi behöver bara sätta in det nya kassettbandet i bandstationen, ställa bandspelaren i beredskap genom att trycka in knapparna READY och RECORD samt kalla på formateringsrutinen genom att trycka på knapparna [1] och [■] på kalkylatorn.

Startkommando ger vi på bandspelarknappen märkt WRITE ON TAPE. Automatiskt kommer nu bandets första fil att bli uppmärkt - dvs pulser inspelas på bandet som indentifie-

rar block nr 1 t o m block nr 14. Varje block upptar tillräcklig plats på bandet för inspelning av hela programminnet eller samtliga 12 minnesregister.

Vi kan upprepa formateringsrutinen för efterföljande filer på bandet. Ett standardband rymmer 12 filer per sida.

Bandets kapacitet är tydligen vid lagring av program:

$$\underbrace{2}_{\text{sidor}} \times \underbrace{12}_{\text{filer}} \times \underbrace{14}_{\text{block}} \times \underbrace{160}_{\text{programsteg}} = 53.760 \text{ programsteg } (= 430.080 \text{ bit})$$

Vid lagring av enbart minnesregister är kapaciteten:

$$\underbrace{2}_{\text{sidor}} \times \underbrace{12}_{\text{filer}} \times \underbrace{14}_{\text{block}} \times \underbrace{12}_{\text{tal}} = 4.032 \text{ trettonsiffriga tal } (= 258.048 \text{ bit})$$

Med den ovan visade enkla formateringsrutinen har bandet försetts med "fack" i vilka vi nu kan lagra informationen, fig 2.14.

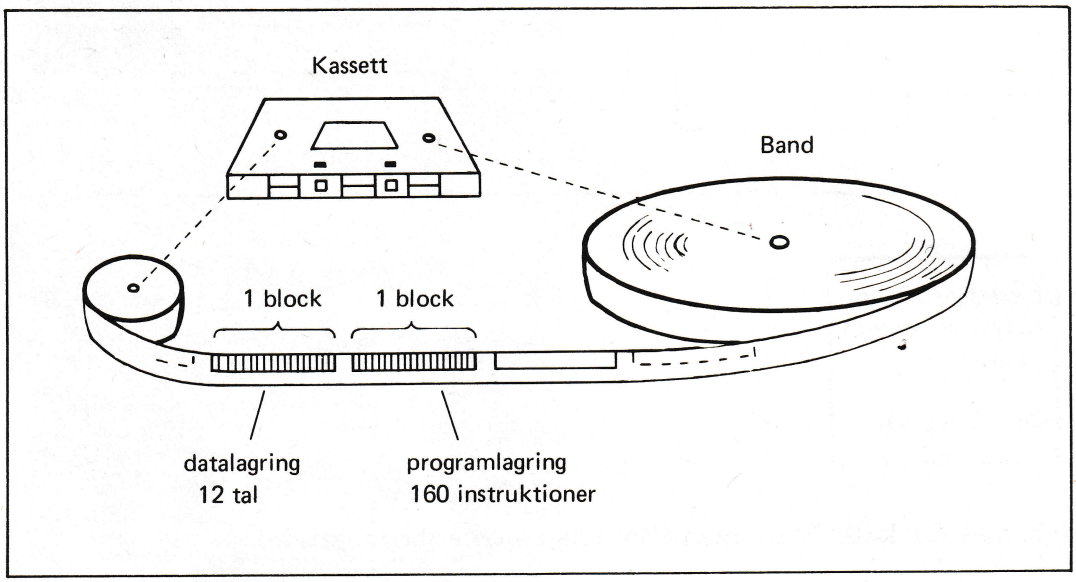


Fig 2.14 Formaterat band för lagring av data eller program.

2.6.2 INSPELNING OCH AVSPELNING

Antag att vårt tidigare program från fig 2.4 (för addering av innehållet i minnesregister 1 och minnesregister 2) finns kvar i kalkylatorns programminne. Hur spelar vi nu in det på vårt formaterade band?

Det är enkelt. Vi spolar tillbaks bandet till början - precis som på en vanlig bandspelare - och ger bandstationen adressen till det block (i fil nr 1) som vi vill ha programmet inspelat på. Detta sker genom att vi på siffertangenterna slår in motsvarande siffra (exempel-

vis 3). Vi gör bandspelaren beredd för inspelning genom att trycka in tangenterna READY and RECORD.

Så fort vi ger startorder med tangenten WRITE börjar bandspelaren leta sig fram till block 3. Där spelar den in programmet varefter den stannar. I indikatorn ser vi nu den adress där programmet spelats in: 3. 0000.

Vi kan lyssna på inspelningen genom att sätta in kassetten i en vanlig ljudbandspelare. Vi hör då dels formateringen - dvs de inspelade pulserna som på jämna mellanrum talar om var informationen i olika block skall lagras - och dels det inspelade programmet. Det senare låter som ett "knatter". Vårt program innehöll ju ca 10 instruktioner, dvs ca 100 bitar, och de ligger inspelade i en följd i block nr 3 i kassettbandets första fil.

Det finns en omkopplare på kalkylatorn med markeringen:

Vi förutsatte ovan att denna omkopplare var ställd i läge PROGRAM. Om denna omkopplare står i läge REGISTERS kommer innehållet i önskat antal av de 12 minnesregistren att spelas in på kassettbandet.

Avspelning av bandet till kalkylatorns programminne eller register tillgår på omvänt sätt med lika enkla procedurer.

Inkopplingen av kassettbandet framgår av blockschemat i fig 2. 13.

Den rutin för inspelning (eller avspelning) av programminne och dataminne som vi ovan beskrivit har varit manuell. Det fina med vår kalkylator 326 är emellertid att dessa rutiner kan utföras på order av programminnet (dvs programstyrda) - förutsatt att vi programmerat kalkylatorn lämpligt.

I instruktionslistan (fig 2. 6) ser vi instruktionerna

040 write program
340 write data
041 read tape
140 preblock (formatering)

Vi kan därmed skriva program som successivt spelar in resultaten av beräkningarna på önskad plats i kassettbandet och som vid behov hämtar nya program (eller data) från givna block i kassettbandet. Det finns kassettband i form av en slinga (endless tape) som innehåller en fil om 14 block. Med ett sådant band kan man återkomma till tidigare block trots att Compucorps bandstation ej kan "backas" med programstyrning.

2. 6. 3 ETT EXEMPEL PÅ ANVÄNDNINGEN AV EN BANDSTATION

Låt oss skriva ett program till vår kalkylator som först hämtar det nyss inspelade adderingsprogrammet från block 3 och som därefter verkställer adderingen av talen i minnesregister 1 och 2 och som slutligen spelar in register 1 - 3 på kassettbandets block 4.

För att under programkontroll kalla in det tidigare programmet från block 3 kan vi göra följande program (fig 2. 15):

Instruktion	PC	Instruktionskod	Kommentar
RESET	001	036	nollställ A- och B-register
3	002	003	adress (anger blocknummer)
READ FROM TAPE	003	041	läs från tape
START STOP	004	033	stopp

Fig 2.15 Exempel på instruktioner som kallar in ett program från bandstationen.

För inspelning av resultatet i minnesregister 1-3 på block fyra måste det tidigare programmet i block 3 (enligt fig 2.4) kompletteras med fyra instruktioner (vilket lätt kan göras med INSERT-tangenten på kalkylatorn). Fig 2.16 visar hela programmet som nu omfattar 14 steg.

Instruktion	PC	Instruktionskod	Kommentar
RESET	001	036	
RCL$_n$	002	310	
1	003	001	
+	004	021	
RCL$_n$	005	310	Samma som i fig 2.4
2	006	002	
=	007	020	
ST$_n$	008	300	
3	009	003	
CLEAR	010	037	Nollställ A-registret (annars får vi 34 i A-registret efter nästa instruktion)
4	011	004	Lagringsadress (anger blocknummer)
PROGRAM REGISTERS WRITE ON TAPE	012	340	Skriv register på tape
3	013	003	Anger antalet register som skall skrivas (i vårt fall 3)
START STOP	014	033	STOPP

Fig 2.16 Programmet i fig 2.4 kompletterat för inspelning av resultatet på band.

När vi modifierat det tidigare programmet i enlighet med fig 2. 16 spelar vi in det på block 3 på bandkassetten. (PROGRAM-REGISTERS omkopplaren skall därvid stå i läge PROGRAM)

Vi knappar sedan in programmet i fig 2. 15 i kalkylatorns programminne.

Låt oss nu anta att det program som vi just knappat in är slutet på ett omfattande beräkningsprogram varunder två delresultat lagras i kalkylatorns minnesregister 1 och 2. För att slutföra beräkningarna inkallas i slutet av detta program ett efterföljande program som finns lagrat i block 3 på kassettbandet (fil 1). För överskådlighets skull har vi i fig 2. 15 enbart medtagit de avslutande instruktionerna, dvs de instruktioner som kallar in det efterföljande programmet.

Programmet i block 3 avslutas (som vi visat i fig 2. 16) med inskrivning av register 1-3 på kassettbandets block 4. Därmed har vi lagrat både delresultat (reg 1 och 2) och slutresultatet (reg 3).

Låt oss köra programserien! Men först måste vi lagra de fingerade delresultaten i register 1 och register 2. Hur det går till har vi tidigare i detalj beskrivit på sidan 30.

Låt oss som exempel lagra talen 4000 och 711 i register 1 resp 2.

RESET 4 0 0 0 ST_n 1

RESET 7 1 1 ST_n 2 RESET

Sedan vi spolat tillbaks bandet till startläget (fil 1) kan vi nu "testa" vårt program. Låt oss starta programmet och köra! Det enda som behövs är att trycka in tangenterna RECORD och READY på bandspelaren samt därefter ge startorder med ett kort tryck på startknappen.

START STOP

Via programstyrningen kallas nu programmet i block 3 in och börjar automatiskt exekveras. Programmet beräknar en summa och sätter detta resultat i minnesregister nr 3. Därefter inskrives resultatet - fortfarande helt under programkontroll - på block 4.

Vi kan göra en flödesplan över förloppet, fig 2. 17.

Om vi nu slår av matningsspänningen till kalkylatorn så nollställs alla register, men resultatet finns lagrat på bandstationen.

Vi slår på kalkylatorn igen, backar bandet och ger läsorder.

4

READ FROM TAPE

Vi har därmed fått resultatet i register 3. Låt oss se efter!

RCL$_n$

3

Faktiskt står det 4711. 0000 i kalkylatorns indikator.

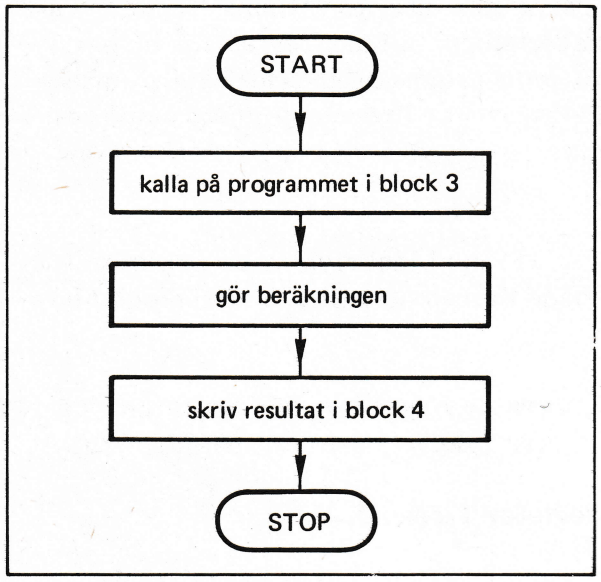


Fig 2.17 Flödesplan för vårt demonstrationsprogram med in- och utmatning via bandstation.

2.7 Några viktiga begrepp

Kalkylatorn som vi blivit bekanta med i det föregående har exemplifierat en rad väsentliga begrepp och funktioner som vi skall återkomma till i samband med minidatorn.

Begrepp	I vårt exempel
minne	ett dataminne och ett programminne
program	följd av instruktioner
programstyrning	vid läsning från och skrivning till bandstation
programmerbar	vi gör programmen själva
instruktioner	instruktionslistan i fig 2. 6
RAM	skriv- läsminne
ROM	läsminne
ord	ett antal bitar
ordlängd	13[x)] decimala siffror i minnesregistren och 2 1/2 oktala siffror i programminnet
minnesorganisation	160 ord om 8 bitar (antal ord där varje ord omfattar givet antal bitar)
data	innehållet i minnesposition
adress	anger ett ords plats i minnet eller ett blocks placering på bandet
programkontroll	läs- och skrivkommando från programminnet
I/O eller in-utenhet kringutrustning	bandstation

Vi har nu tränat in de viktigaste begreppen vid elektronisk hantering av data och vi är nu mogna att ge oss i kast med minidatorn ALPHA LSI.

---

x) 13-siffrig mantissa och 2-siffrig exponent

3. ALPHA LSI

Innehåll

3. ALPHA LSI 51

3.1 En första presentation 52
- 3.1.1 Byggblocken 52
- 3.1.2 Ett förenklat blockschema 55
- 3.1.3 Ready console 58
- 3.1.4 Kontrollpanelen 59
- 3.1.5 Vårt första program 62
- 3.1.6 Vad har vi lärt oss ? 70

3.2 Något om elektroniken 72
- 3.2.1 Räkneenheten 72
- 3.2.2 Att lagra data 78
- 3.2.3 Att förflytta data 89
- 3.2.4 Styrenheten 99
- 3.2.5 Exekveringen 111
- 3.2.6 Adressering av en operand 118
- 3.2.7 Hopp 121
- 3.2.8 Kraftförsörjningen 122

3.3 Input/Output 126
- 3.3.1 TTY:n och dess funktioner 126
- 3.3.2 TTY:n som sändare och mottagare 132
- 3.3.3 ASCII 133
- 3.3.4 TTY-interfacet 136

3.4 Software support 150
- 3.4.1 Assemblerprogrammering 151
- 3.4.2 Högnivåspråk 168
- 3.4.3 Operativsystem 168

3.5 Kassaprogrammet - ett avslutande exempel 171
- 3.5.1 Önskad funktion 171
- 3.5.2 Flödesplanen 173
- 3.5.3 Assemblerprogrammet 176
- 3.5.4 Några detaljer i assemblerprogrammet 184
- 3.5.5 Från källkod till kassaprogram 188
- 3.5.6 Kassaprogram i BASIC 188

3. ALPHA LSI

Vi ska nu ge oss i kast med minidatorn ALPHA LSI. För att inte framställningen ska försvinna bland mängder av termer och detaljer ska vi börja med en förenklad bild av vår minidator och vi ska redan från början utföra en av de beräkningar som vi tidigare gjort både i räknedosan 10R och kalkylatorn 326.

När vi sett att ALPHA LSI fungerar, ska vi ge oss litet närmare in på elektroniken. Alla kretsarna på det stora CPU-kortet ser ju så lika ut i sina svarta DIL-kapslar, men de fullgör ändå en rad olika funktioner.

En speciellt intressant del av datorn är styrenheten. Den får sina order från instruktionsregistret och vi ska följa arbetsförloppet (en cykel) vid verkställandet av en enkel instruktion.

En dators uppgift är inte att stå och räkna för sig själv. Den måste ju hämta sina uppgifter och avge resultatet till omgivningen. Det sker via I/O-enheter, som det kallas på datorspråk (I/O = input/output). Vi har bara en I/O-enhet till vår dator och det är en fjärrskrivmaskin (teletypewriter) eller TTY som den vanligen kallas. TTY:n är en gammal och beprövad apparat, men den används fortfarande flitigast av alla "terminaler" för minidatorsystem. Vi ska se hur den arbetar och styrs från datorn.

När vi kommit så här långt kan vi börja avlasta oss detaljarbetet. Mycket av det detaljarbete vi lagt ner på programmeringen för våra enkla demonstrationsprogram borde ha kunnat utföras betydligt enklare om vi tagit datorn själv till hjälp. Vi ska bekanta oss med några sådana hjälpmedel eller "software supports".

De flesta som använder datorer bryr sig knappast om vad som döljer sig i datorns inre. De använder något av de högnivåspråk som medger att man nästan talar i "klartext" till datorn. Basic är ett sådant språk. Vi ska avsluta vår genomgång av ALPHA LSI med att ladda in en "Basic-tolk" och köra ett program i Basic.

3.1 En första presentation

Fig 3.1 visar den utrustning vi ska ägna oss åt. Den består av en dator typ ALPHA LSI 2 (tillverkad av Computer Automation i Orange County strax söder om Los Angeles i USA) och en fjärrskrivare eller teletype (typ ASR 33 från Teletype Corporation, Illinois). Både datorn och TTY:n tillverkas i en mängd olika varianter.

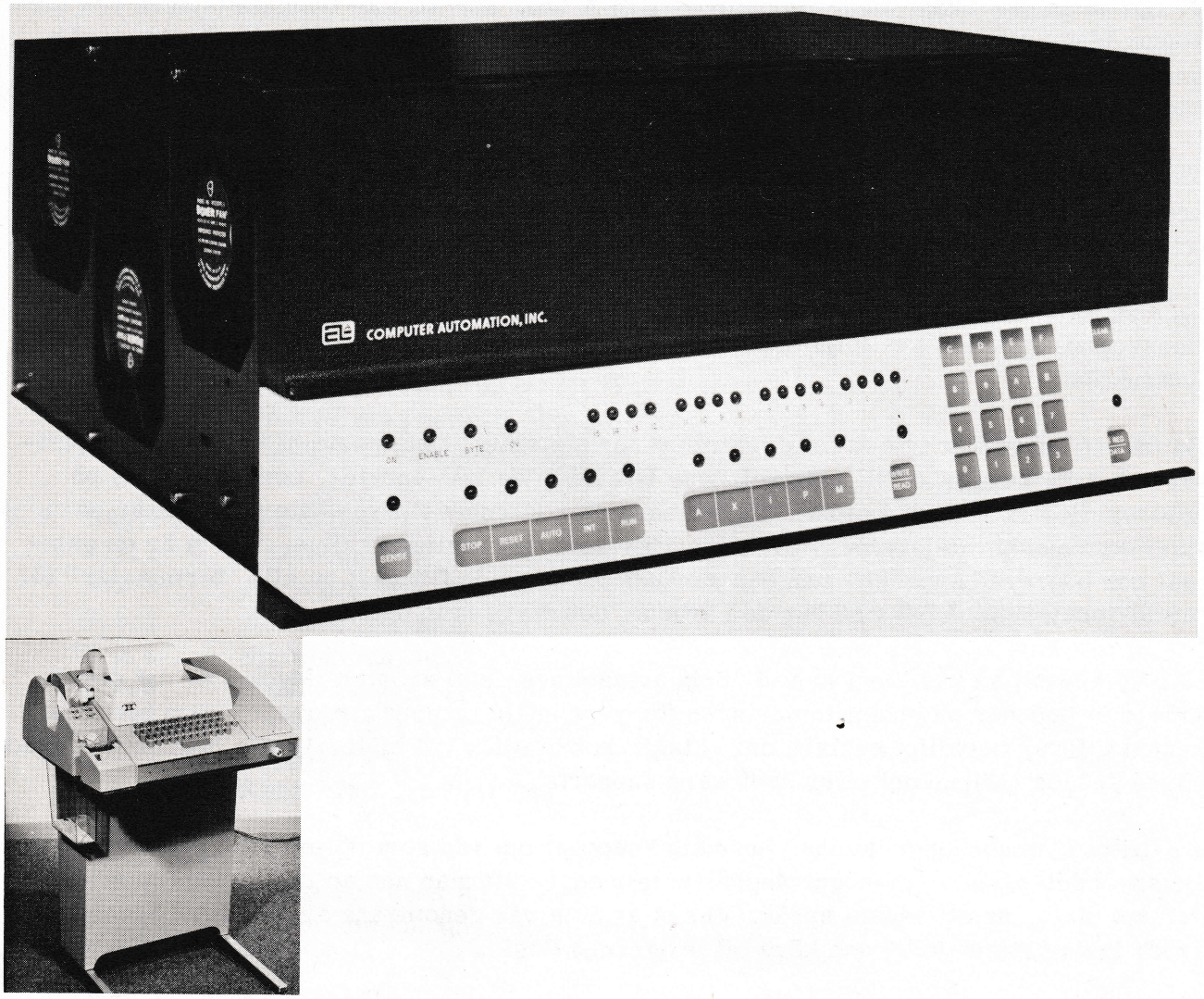


Fig 3.1 Vårt datorsystem, en ALPHA LSI 2 och en teletype, ASR 33.

3.1.1 BYGGBLOCKEN

ALPHA LSI är egentligen ett byggsystem, som är uppbyggt kring den processor (CPU = central processing unit) som vi ser på fig 3.2. Det var först när man på detta sätt kunde bygga upp processorer på ett enda kort, som datorer gick att tillverka till rimliga priser. Med den här tekniken, och dess fortsättning mot alltmer högintegrerade kretsar, kommer datorer snart att bli var mans egendom och hjälpmedel.

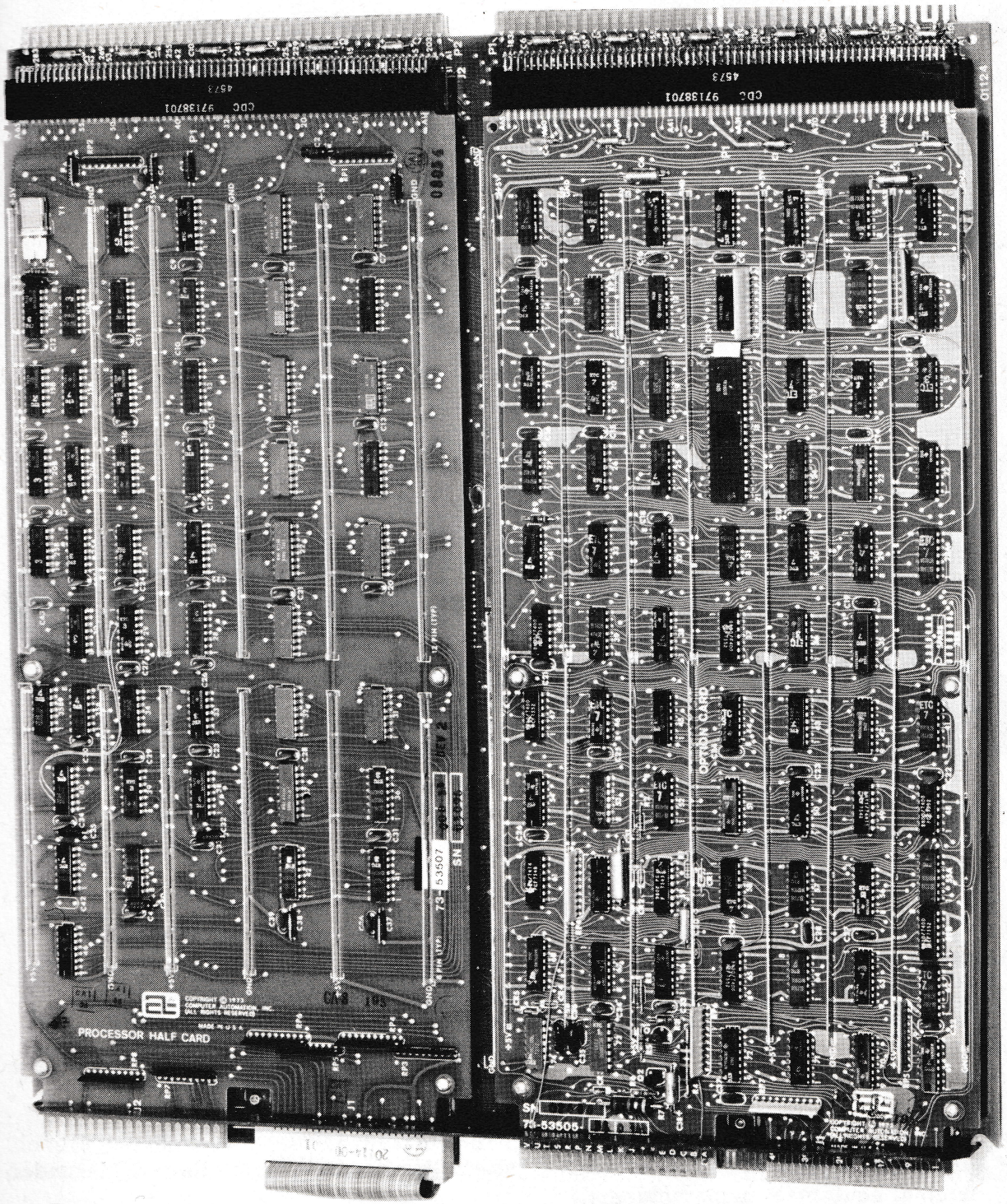


Fig 3.2 CPU:n i ALPHA LSI kallas NAKED MINI, allt på ett kort.

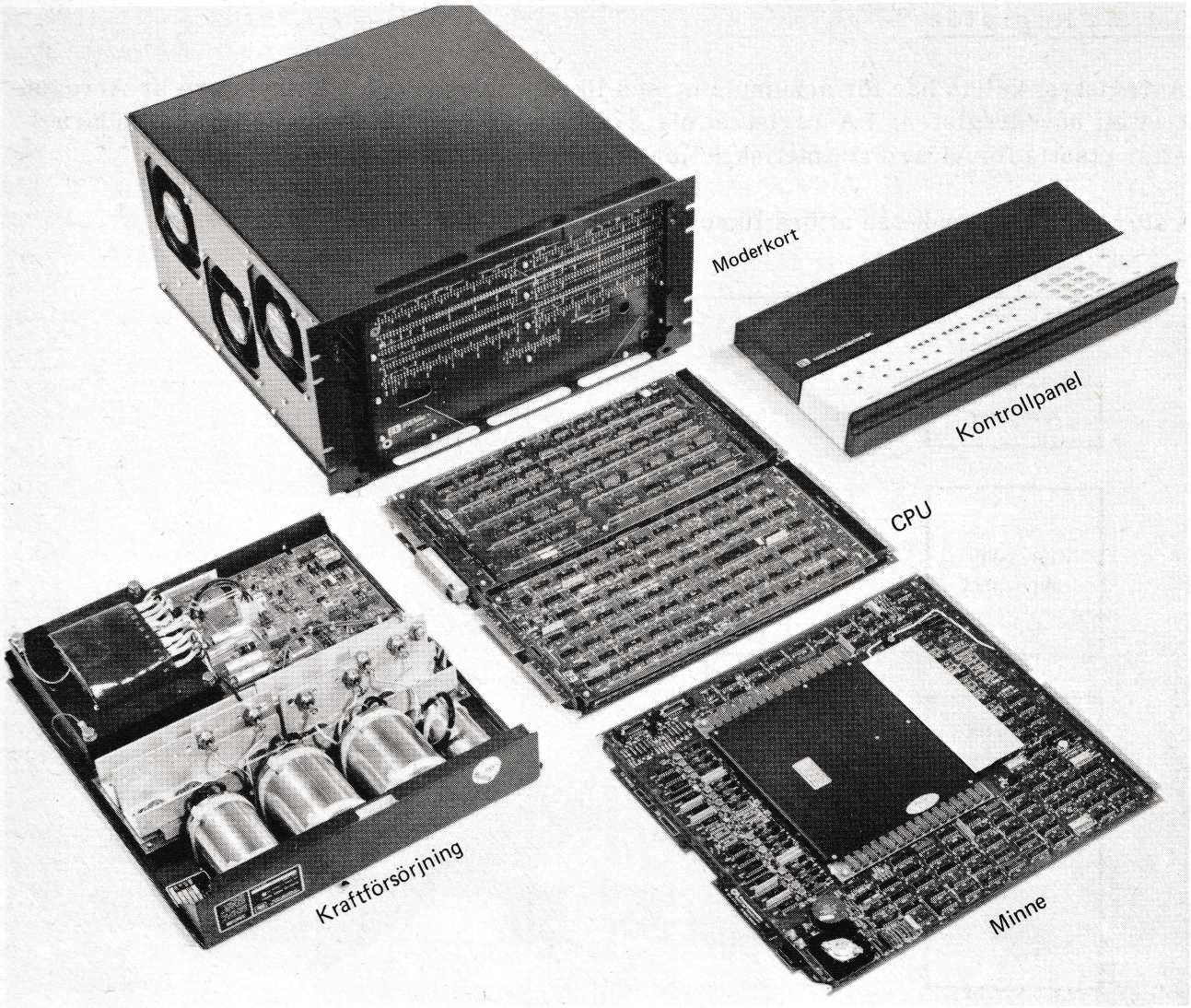


Fig 3.4 ALPHA LSI - ett byggsystem. CPU-kortet i mitten. Kortet med strömförsörjning ligger till vänster och minneskortet längst fram. I lådan ser man moderkortet.

Moderkortet förbinder samtliga kretskort med varandra samt med nätdelen och kontrollpanelen. Det finns plats för ytterligare tre kretskort i lådan och det utrymmet är avsett för dem som vill sätta in större minne eller som vill bygga "interface", dvs elektronikkretsar som anpassar datorn till olika omgivande apparater (exempelvis en numeriskt styrd borrmaskin, ett skivminne eller ett kassaregister).

3.1.2 ETT FÖRENKLAT BLOCKSCHEMA

Fig 3.5 visar ett blockschema liknande de enkla scheman som vi tidigare använt för att beskriva räknedosan och kalkylatorn (fig 1.11 och fig 2.2). Vi ska nedan beskriva de viktigaste byggblocken.

3.1.2.1 Register

A-registret kallas här för ackumulator och liksom i dosan och i kalkylatorn är A-registret ett huvudregister. I A-registret placeras alla delresultat innan de forslas vidare eller utsätts för vidare aritmetisk (eller logisk) bearbetning.

I styr- och räkneenheten utförs liksom tidigare alla operationer.

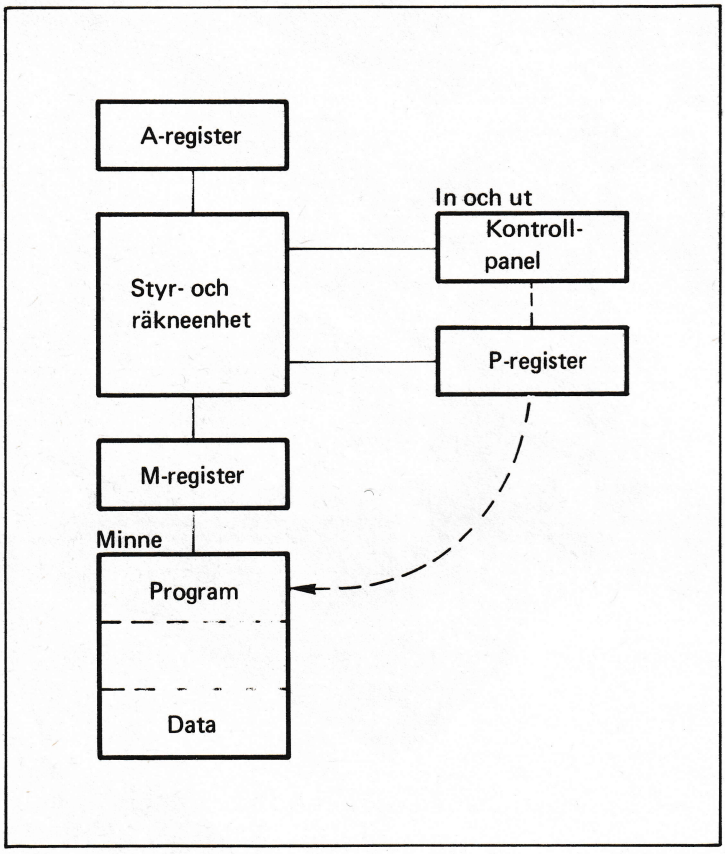


Fig 3.5 Ett förenklat blockschema.

Istället för räknedosans B-register (eller kalkylatorns B-register och minnes-register) har ALPHA LSI ett M-register. M-registret visar innehållet i en av cellerna (dvs i ett ord) i det stora primärminnet. Varje cell i minnet har en adress. När man vill ha reda på innehållet i en cell ger man minnet den aktuella adressen varvid minnet svarar med att kopiera den adresserade cellen till M-registret.

P-registret (dvs programräknaren) används normalt för att peka ut (dvs adressera) de celler i minnet som innehåller programmet.

3.1.2.2 Minne

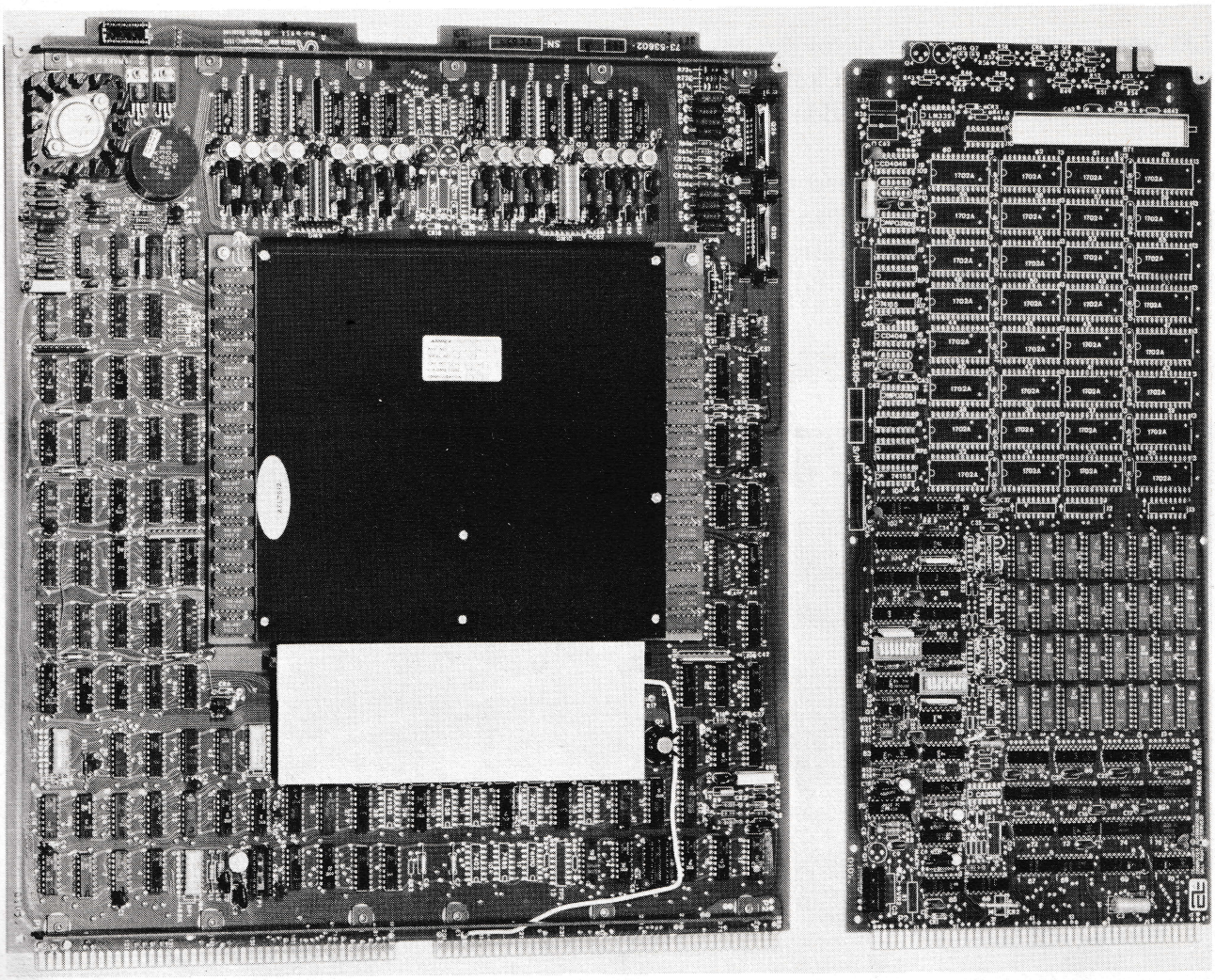
ALPHA LSI har ett kärnminne som kan byggas ut i moduler om 4 x 1024 ord (om varde-

ra 16 bitar). 1024 ord brukar kallas 1 kiloord och "kilo" har alltså en litet speciell tolkning i samband med minnen.

Vårt datorsystem i fig 3.1 har ett minne om "8 k" (dvs 8192 ord). Ordlängden är (som hos de flesta minidatorer) 16 bitar och minnet innehåller alltså 131.072 bitar. Varje bit utgörs av en ferritkärna som kan magnetiseras i endera av två riktningar (dvs kan lagra en bit). Ett kärnminne är icke "volatilt" dvs det tappar inte minnet om matningsspänningen faller ifrån.

Det finns, som ett alternativ till kärnminnen, kretskort med halvledarminnen. Bitarna består då i princip av vippor som står stabilt i inställt läge så länge matningsspänningen ligger kvar. Man brukar kalla datorns kärn- eller halvledarminne för primärminne eftersom man ofta har yttre sekundärminnen i form av band, trummor eller skivor.

Fig 3.6 visar kretskort med kärn- och halvledarminnen till ALPHA LSI.



Kärnminne Halvledarminne

Fig 3.6 Kärnminne till vänster och halvledarminne till ALPHA LSI.

3.1.2.3 Kontrollpanel

Kontrollpanelen för vår dator är egentligen inte avsedd att användas "för vardagsbruk". Den är till för uppstartning och service. Vi ska emellertid använda kontrollpanelen både som tangentbord (dvs för input) och som indikator (dvs för output) när vi nu ska pröva datorns funktion. Eftersom panelen inte är avsedd för vardagsbruk kan den urkopplas. Den är inte lika bekväm att använda som räkndosans och kalkylatorns tangenter och decimala indikatorer.

Blockschemat i fig 3.5 visar enbart en del av de många blocken i ALPHA LSI. I själva verket innehåller ALPHA LSI ytterligare fyra register. Men för våra närmaste behov räcker schemat i fig 3.5 mycket väl.

3.1.3 READY CONSOLE

Låt oss nu pröva om vår dator fungerar. Se till att nätsladden är inkopplad (TTY:n kan också vara inkopplad men vi kommer inte att använda den än). Huvudströmställaren (POWER ON) sitter strax över nätsladden på datorns baksida (fig 3.7).

Koppla in kontrollpanelen med strömställaren (ENABLE CONSOLE) som sitter gömd i panelens högra kant (framifrån sett)! När nätspänningen är tillslagen lyser den vänstra översta lysdioden (ON) och när kontrollpanelen är inkopplad lyser dioden närmast till höger därom (ENABLE). Nu bör några av lysdioderna i kontrollpanelen lysa vackert röda. Fig 3.8.

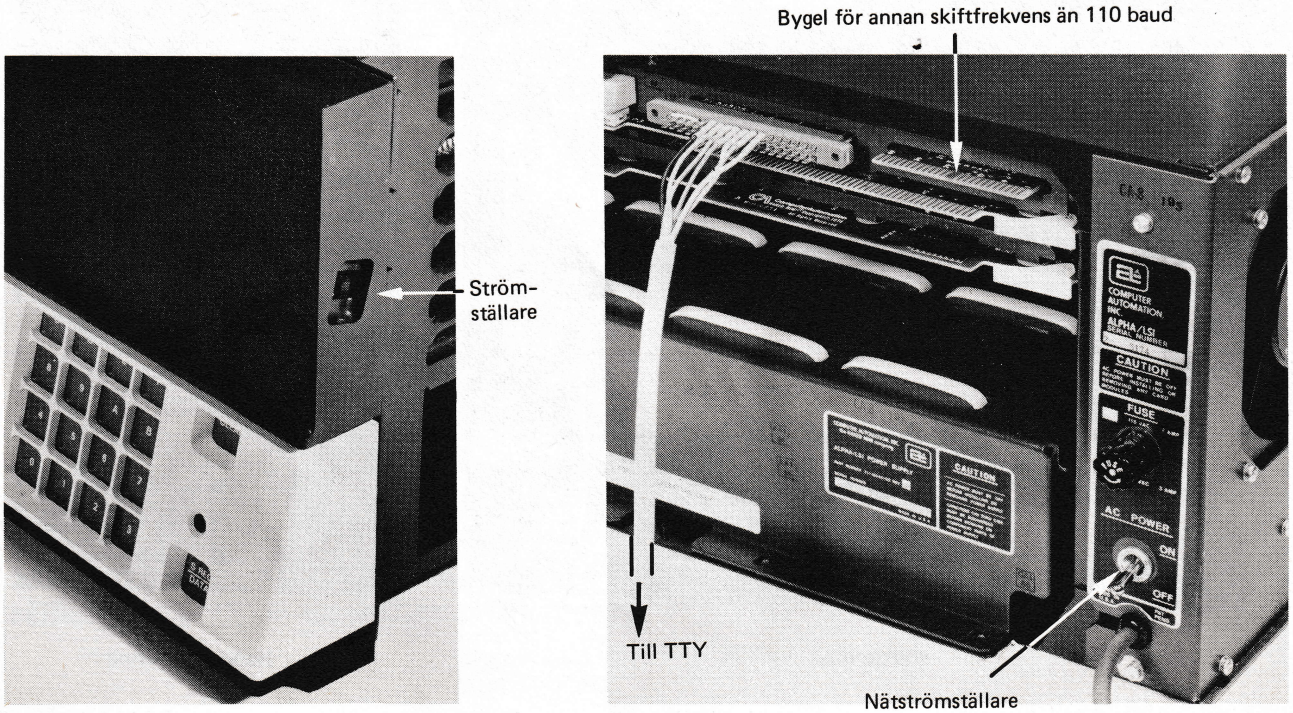


Fig 3.7 Två väsentliga strömställare på ALPHA LSI.

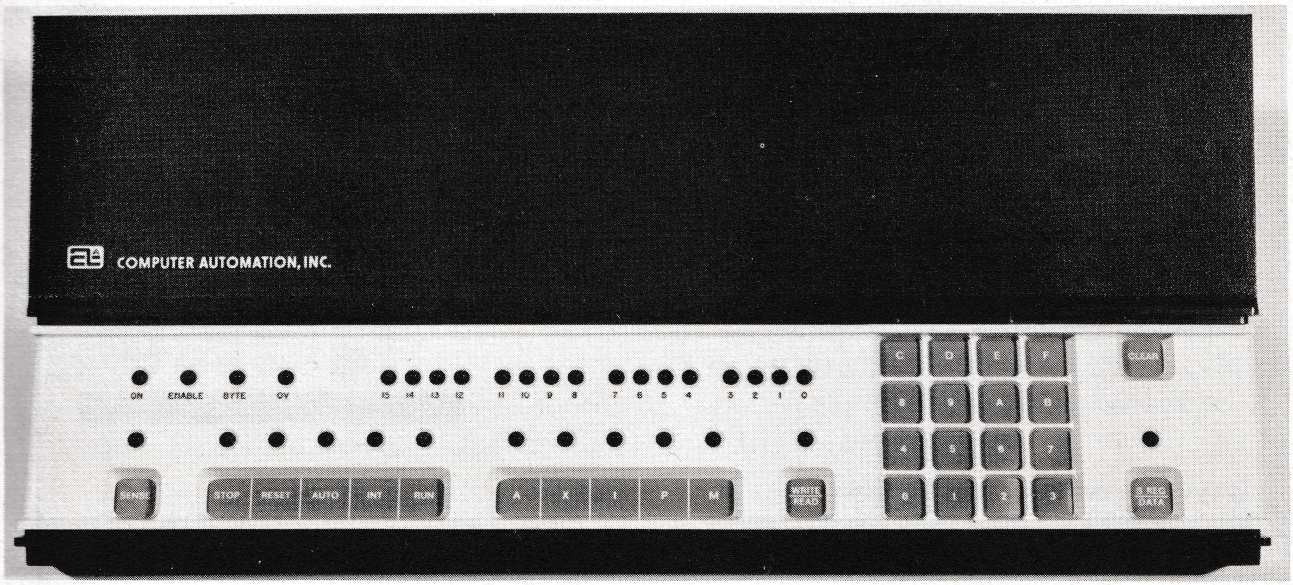


Fig 3.8 Kontrollpanelen på ALPHA LSI.

Datorn är nu litet nyvaken och för att få den att "samla sig" trycker vi på STOP och på RESET. Om kärnminnet innehåller något man är rädd om, t ex BASIC-interpretatorn, är det lämpligt att hålla STOP intryckt vid POWER-ON. Nu står ALPHA LSI snällt still och avvaktar våra vidare order, men innan vi trycker på fler knappar ska vi kort beskriva panelen.

READY CONSOLE kallas den startrutin som vi utfört ovan. Den kan vi alltid använda när vi startar vår dator med hjälp av kontrollpanelen.[x)]

3.1.4 KONTROLLPANELEN

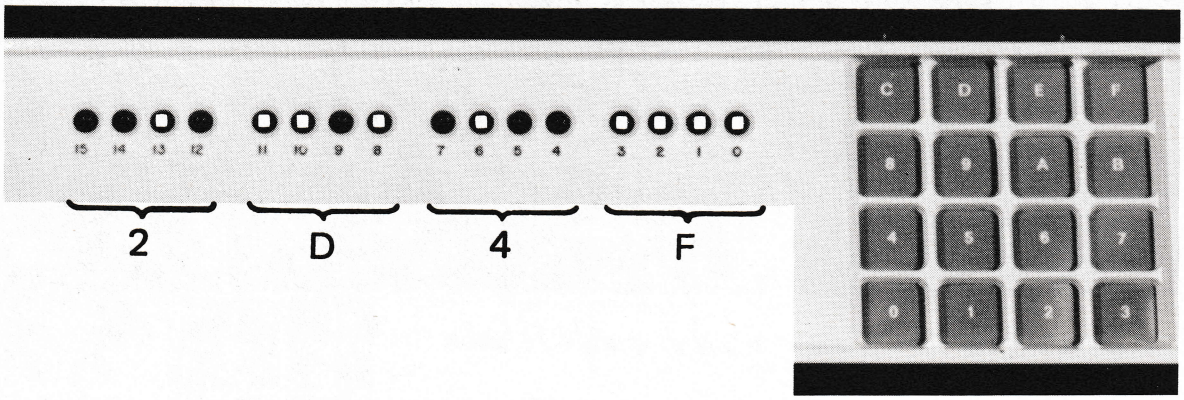
Varje minidator med självaktning har en kontrollpanel med en mängd knappar och lampor. ALPHA LSI är härvid inget undantag, men den är trots allt ganska enkel att vänja sig vid. Kontrollpanelen visas i fig 3.8 och vi ska gå igenom de viktigaste funktionerna.

Överst (i panelens mitt) ser vi 16 lysdioder ("lampor"). De bildar ett binärt display (eller indikator) och är givetvis inte så överskådliga att avläsa som våra tidigare decimala display på räknedosan och på kalkylatorn. Knappsatsen till höger har 16 hexadecimala knappar. Det känns i början litet ovanligt att hantera hexidecimala tangenter jämfört med dosans och kalkylatorns decimala siffertangenter.

Tryck in en av de 16 tangenterna! Vi får direkt motsvarande binärtal i lysande form i de fyra lysdioderna längst till höger i indikatorn. Fig 3.9.

---

x) Vi bortser i denna beskrivning från den allra enklaste starten vid BASIC vilken tillgå på följande sätt: Starta först TTY:n och sedan datorn .... och BASIC startar!!



a) Lysdioder och hexadecimal knappsats.

○○○○ 0	●○○○ 8	
○○○● 1	●○○● 9	
○○●○ 2	●○●○ A	(10)
○○●● 3	●○●● B	(11)
○●○○ 4	●●○○ C	(12)
○●○● 5	●●○● D	(13)
○●●○ 6	●●●○ E	(14)
○●●● 7	●●●● F	(15)

b) Hexadecimala koden.

Fig 3.9 a. Binär indikator och hexadecimal knappsats.
b. Binär och hexadecimal kod.

Vi kan skifta in tal (från höger) i indikatorn om lampan över S REG/DATA-knappen till höger om knappsatsen är släckt (tryck i annat fall på S REG/DATA-knappen så slocknar den).

Pröva med olika hextal och träna på att avläsa dem på indikatorn. Tack vare den hexadecimala knappsatsen fyller vi indikatorn helt med enbart fyra tangentnedtryckningar (jämfört med 16 vid binära knappar). Efter litet träning arbetar man ganska snabbt med knappsatsen.

3.1.4.1 Kontrollpanelens register

Kontrollpanelen innehåller två register:

- ett 16 bitars DATA-register (console data register)
- ett 4 bitars SENSE-register (console sense register), förkortat S REG

Då S REG/DATA-lampan är släckt så är dataregistret inkopplat.

Man kan avläsa datorns olika interna register genom att överföra deras innehåll till kontrollpanelens dataregister. Vid avläsning skall WRITE/READ-lampan (omedelbart till vänster om knappsatsen) vara släckt. Man kan skriva in tal i de interna registren från kontrollpanelens dataregister. Då ska WRITE/READ-lampan vara tänd.

3.1.4.2 Inskrivning och avläsning

Om WRITE/READ-lampan är släckt kan man till kontrollpanelens dataregister kopiera innehållet hos fem av de sju interna registren i ALPHA LSI, nämligen A-, X-, I-, M- och P-registren. Kopieringen sker direkt när man trycker på någon av knapparna A, X, I, P eller M (omedelbart till vänster om READ/WRITE-knappen).

Om WRITE/READ-lampan lyser kan de interna registren laddas. Med knappsatsen laddar man därvid först panelens dataregister. Dataregistret kopieras sedan på ett av de interna registren, när man trycker på någon av knapparna A, X, I, M eller P.

Vi är för närvarande endast intresserade av tre av dessa register:

A-registret, dvs ackumulatorn
P-registret, dvs programräknaren
M-registret, dvs det register som slussar information till och från minnet.

Låt oss kontrollera innehållet i dem! Se först till att WRITE/READ-lampan är släckt. Sedan är det bara att trycka in någon av knapparna med beteckningarna A, P eller M så får vi direkt innehållet i dessa register i indikatorn.

M-registret skiljer sig från de övriga registren. Det visar innehållet i den minnescell som utpekats av P-registret - men vi skall strax återkomma till M-registret och minnet.

Att skriva in talet FFFF i ackumulatorn går till så här.

Kontrollera först att datorn står still (STOP-lampan lyser) och att dataregistret är inkopplat (S REG/DATA-lampan är släckt).

1. Fyra tryck på F-tangenten (varefter samtliga 16 lysdioder skall lysa)
2. Se till att WRITE/READ-lampan lyser
3. Tryck på A-knappen

Att skriva in talet 0000 i programräknaren sker enkelt med följande operationer.

1. Tryck på CLEAR-knappen längst upp till höger (nollställer dataregistret med ett enda tryck)
2. Se till att WRITE/READ-lampan lyser
3. Tryck på P-knappen

Under rubriken CONSOLE i "Programming reference manual" för ALPHA LSI finns en utförlig beskrivning av kontrollpanelens alla detaljer.

3.1.5 VÅRT FÖRSTA PROGRAM

Räknedosan gav vi order via instruktionstangenterna [+], [−], [×] eller [÷]. I kalkylatorn hade vi ett stort antal instruktionstangenter (eller instruktioner) att välja på. ALPHA LSI har däremot inga instruktionstangenter och hur ger vi då datorn våra instruktioner?

När vi skrev ett program i vår Compucorp-kalkylator behövde vi bara trycka in en följd av instruktionstangenter och vi fick då automatiskt motsvarande program lagrat i kalkylatorns programminne. Men ALPHA LSI har inget speciellt programminne. Hur ska vi då kunna ge vår dator ett arbetsprogram?

Svaren på ovanstående frågor finner vi om vi studerar "instruktionslistan" och "minnet" hos ALPHA LSI.

3.1.5.1 Instruktionslistan

En dator har inga instruktionstangenter men däremot en instruktionslista. En instruktionslista är en uppställning av alla de sifferkombinationer (ord) som datorn kan tolka som en instruktion. ALPHA LSI har en lista som omfattar 168 olika instruktioner (och många av dessa kan skrivas i ett stort antal varianter). Både dator- och mikroprocessorfabrikanter brukar framhålla hur många och praktiskt användbara instruktioner deras maskiner har.

I vår tidigare kalkylator omfattade varje instruktion tre oktala siffror och instruktionslistan omfattade 183 instruktioner (fig 2.6). Vår dator arbetar med 16-bitars ordlängd och de flesta instruktionerna är därför på 16 bitar.

Vi ska strax göra ett program som adderar innehållet från två celler i minnet och placerar summan i en tredje cell. Det är alltså samma program som vi tidigare utfört med kalkylatorn (fig 2.4). För detta ändamål behöver vi fyra instruktioner, som man efter en stunds letande hittar i programmeringsmanualen (fig 3.10). De har beteckningarna LDA, ADD, STA och HLT. Dessa beteckningar är så valda att man enkelt skall komma ihåg funktionerna (MEMOsymbol, dvs hjälper oss att minnas).

De tre första instruktionerna i fig 3.10 kallas minnesrefererande (memory reference instructions). De avser ju operationer på tal som är lagrade eller skall lagras i minnet. Sådana instruktioner har ett speciellt "format", dvs de sexton bitarna i instruktionen är indelade i grupper som har speciell tolkning. Fig 3.11.

De tal vi hanterar (dvs hämtar ur minnet och adderar etc) kallas "operander". Eftersom de är lagrade i minnet så har de en adress, dvs en operandadress. För att spara minnesutrymme låter man minnesrefererande instruktioner som LDA, ADD och STA innehålla både "operationsordern" (exempelvis "addera") och "operandadressen".

Instruktionen ADD (som totalt omfattar 16 bitar) är indelad i två byte[x)]. Den första (dvs

---

x) Ordet byte används ofta för en grupp om 8 bitar. Byte uttalas "bait."

MEMO-symbol (mnemonic)	Grundkod (hexadecimal)	Kommentar
LDA	B000	Ladda A med innehållet i en minnescell
ADD	8800	addera talet i A till talet i en minnescell och sätt summan i A
STA	9800	lagra A:s innehåll i en minnescell
HLT	0800	STOPP

Fig 3.10 Fyra instruktioner ur instruktionslistan till ALPHA LSI. (A = ackumulator).

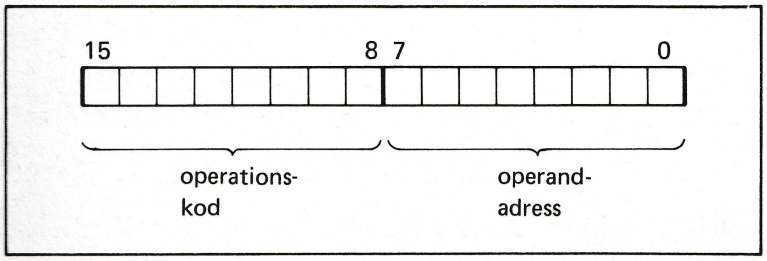


Fig 3.11 Instruktion av format 1: Minnesrefererande instruktion.

pos 15-8) anger operationsordern med hexadecimala koden 88 (eller :88 som det ofta skrivs i ALPHA LSI:s manualer). Den andra byten (dvs pos 7 - 0) anger operandadress.

En lista av instruktioner som är skrivna exakt enligt det format som datorn kan tolka bildar ett program. Vi skall strax återkomma till detta.

3.1.5.2 Minnet

Datorer är "allmänbruksmaskiner" som skall kunna användas för vitt skilda uppgifter. Ibland hanteras stora datamängder med korta program och vid andra tillfällen utförs komplicerade beräkningar som kräver stort programminne men som endast behöver mindre dataminne.

Datorer innehåller därför som regel ett enda stort primärminne och detta utnyttjas både för programlagring och datalagring. Fig 3.12.

Minnet i vår dator i fig 3.1 innehåller 8 kord och dessa ord har adresserna: 0000 till och med: 1FFF (hexadecimalt). Program och data kan vi placera godtyckligt i minnet. Programmet brukar som regel placeras i en sammanhängande följd. Det är programräknaren P som håller reda på instruktionsadresserna. P räknas automatiskt fram ett steg för varje ny instruktion som hämtas och utförs. På detta sätt exekveras instruktionerna automatiskt om de är placerade i en följd i minnet. Före starten ställer man in programräknaren P på startadressen och på startkommando (RUN) arbetar sedan datorn igenom instruktion efter instruktion till den slutligen träffar på en stoppinstruktion.

Vi har här inte berört "hoppinstruktionerna" men vi återkomma till dem senare.

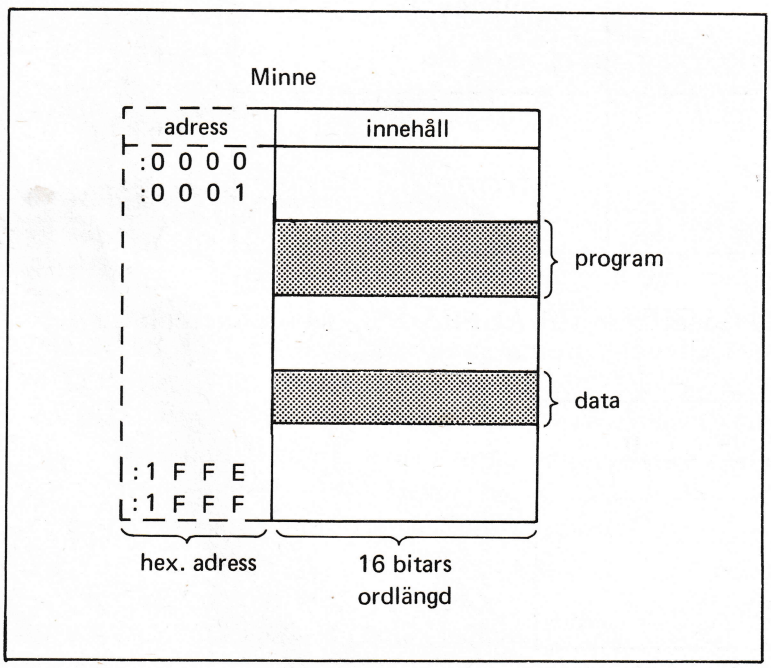


Fig 3.12 Ett minne omfattande 8 kord. ($1FFF_{16} = 8191_{10}$)

3.1.5.1 Programmet

Det "klassiska" demonstrationsprogrammet hämtar två tal från olika celler i minnet, adderar dem och sätter summan i en tredje cell i minnet. Vi ska börja med ett sådant demonstrationsprogram.

Vi kan som sagt lagra program och data på godtycklig plats i minnet, men vi väljer nu

att lagra programmet med början i cellen med adress 0. Vi ska strax se att det åtgår fyra instruktioner och vi lagrar därför de två tal som skall adderas i cellerna 4 och 5 (dvs med adresserna 0004 och 0005). Cell 6 reserverar vi för lagring av resultatet.

Programmet kan skrivas så som visas i fig 3.13. Vi använder memosymboler (mnemonic) för att lättare känna igen funktionerna hos de enskilda instruktionerna.

Programsteg nr	Instruktion operationskod	adress
1	LDA	4
2	ADD	5
3	STA	6
4	HLT	

Fig 3.13 Ett enkelt program.

Programmet i fig 3.13 kan tolkas i ord på följande sätt:

LDA 4: Ladda ackumulatorn A med innehållet i minnescell 4 (dvs med adressen 4).

ADD 5: Addera innehållet i A med innehållet i minnescell 5 och sätt därefter summan i A.

STA 6: Lagra innehållet i A i minnescell 6.

HLT: Stopp (eller halt).

Datorn läser inte memosymboler utan vill ha varje instruktion som ett 16-bitars binärtal.

Grundkoden för LDA är enligt fig 3.10

LDA B0 00

operationskod	operandadress

Sätter vi in operandadressen 4 blir vår första instruktion LDA 4

LDA 4 :B004

B004 (eller :B004) är ett hexadecimalt skrivsätt för en instruktion som innehåller 16 bitar och som kan tolkas av datorn. Den kallas därför maskininstruktion (och är skriven i maskinkod).

På samma sätt får vi instruktionerna

ADD 5 :8805
STA 6 :9806

Haltinstruktionen är ej minnesrefererande och har därför ingen adressdel. Den är alltid

HLT :0800

Det program vi ska ladda in i datorn ser alltså ut på följande sätt:

Hex	15 Binärt 0			
B 0 0 4	1 0 1 1	0 0 0 0	0 0 0 0	0 1 0 0
8 8 0 5	1 0 0 0	1 0 0 0	0 0 0 0	0 1 0 1
9 8 0 6	1 0 0 1	1 0 0 0	0 0 0 0	0 1 1 0
0 8 0 0	0 0 0 0	1 0 0 0	0 0 0 0	0 0 0 0

Vi kan ladda detta program på godtycklig plats i minnet. Var det än läggs in så hämtar programmet talen i cell 4 och cell 5, adderar dem och sätter summan i cell 6.

För enkelhetens skull lägger vi nu programmet i cellerna 0 till och med 3.

Låt oss alltså skriva in följande i ALPHA LSI:s minne:

adress	innehåll	
0	B 0 0 4	
1	8 8 0 5	
2	9 8 0 6	
3	0 8 0 0	
4	0 0 0 3	(talet 3)
5	0 0 0 4	(talet 4)
6	0 0 0 0	(reserverat för summan)

Hur gör vi nu rent praktiskt? Det är ganska enkelt. Gör först READY CONSOLE-rutinen som vi beskrivit tidigare (sid 58).

Principen för att via kontrollpanelen skriva in ett tal i minnet är följande:

- Ställ programräknaren (P) på den önskade adressen.
- Placera talet i kontrollpanelens dataregister.
- Skriv in med ett tryck på M-knappen (WRITE/READ-lampan måste lysa). Om man tittar på RUN-lampan samtidigt som man trycker på M-knappen ser man en blinkning. Det är ett tecken på att inskrivningen är utförd.

För varje inskrivning vi gör, stegas programräknaren fram ett steg. Därför behöver vi bara ställa in första adressen. Knapparnas placering framgår av fig 3.14.

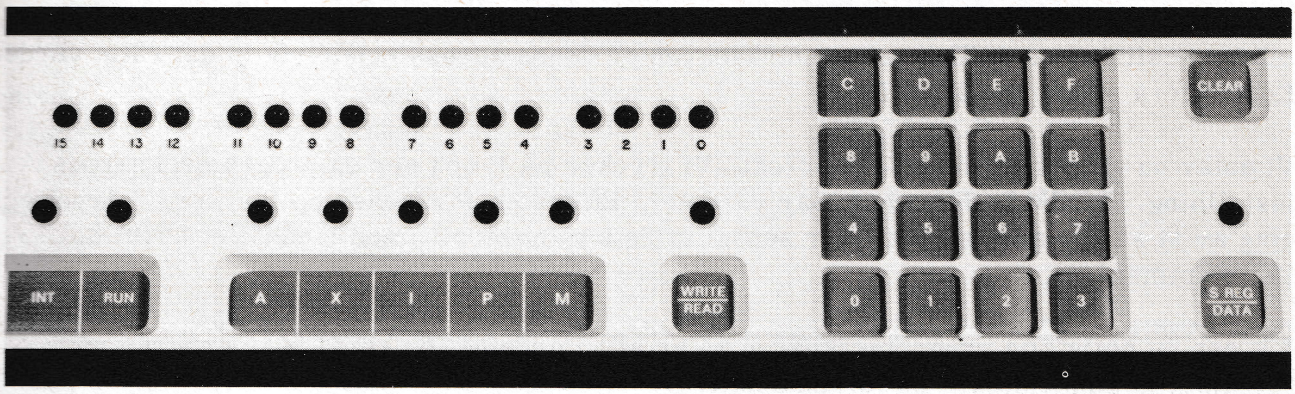


Fig 3.14 Del av kontrollpanelen på ALPHA LSI.

Med WRITE/READ-lampan tänd och S REG/DATA-lampan släckt knappar vi alltså in följande:

CLEAR	Nollställer kontrollpanelens dataregister
P	Kopierar dataregistret till P-registret (dvs 0000 ⇒ P)
B004	Sätter B004 i dataregistret
M	Kopierar dataregistret till M-registret (och därmed till adress P = 0000 i minnet)
8805	etc
M	
9806	
M	
0800	
M	
0003	
M	
0004	
M	
0000	
M	

För att kontrollera att vi skrivit in rätt i minnet läser vi nu av samma minnesceller, genom att återigen ställa programräknaren på noll.

Nu måste vi släcka WRITE/READ-lampan så att panelen läser (tryck på WRITE/READ-knappen så slocknar den!)

Nu är det bara att trycka på M och läsa av. Förhoppningsvis får vi då följande:

Tryck på	I indikatorn står	
M	B004	(innehåll i minnescell 0000)
M	8805	(innehåll i minnescell 0001)
M	9806	(innehåll i minnescell 0002)
M	0800	(innehåll i minnescell 0003)
M	0003	(innehåll i minnescell 0004)
M	0004	(innehåll i minnescell 0005)
M	0000	(innehåll i minnescell 0006)

Kontrollera innehållet noga! Det är så lätt att slå in ett felaktigt värde. Är det fel i en minnesposition, så ställ in programräknaren på denna adress och skriv in det rätta värdet. Kontrollera sedan en gång till!

Vi antar nu att programmet är korrekt. Vi trycker på P och ser då programräknarens inställning. Lysdiodindikatorn visar värdet 7. Eftersom P räknas upp för varje intryckning av M så var adressen till vår senast utlästa minnescell (med innehållet 0000) 6. Det stämmer!

Låt oss nu nollställa programräknaren så är allt i ordning för körning av programmet. P nollställs som tidigare på följande sätt:

- tryck på CLEAR
- tänd WRITE/READ - lampan
- tryck på P

3.1.5.4 Fyra arbetsmoder

Hittills har vi enbart arbetat med att ladda och avläsa register och minnesceller. Nu ska vi ge oss i kast med datorns start och stoppkommandon.

Till vänster om panelens registerknappar (A, X, I, P och M) ser vi en rad knappar märkta så som framgår av fig 3.15.

STOP- och RESET-knapparna har vi redan tidigare använt vid READY-CONSOLE rutinen.

RESET-knappen ska normalt inte behöva användas. Det är bara när datorn kommer in i en slinga (loop) och inte vill stanna, som man behöver tillgripa RESET (dvs omedelbar nollställning av ett antal kontrollvippor).

Med STOP-knappen stoppar man datorns arbete, efter det att pågående instruktion slutförts.

Startknappen heter RUN. RUN- och STOP-lamporna kan tändas oberoende av varandra och man får på detta sätt fyra arbetsmoder.

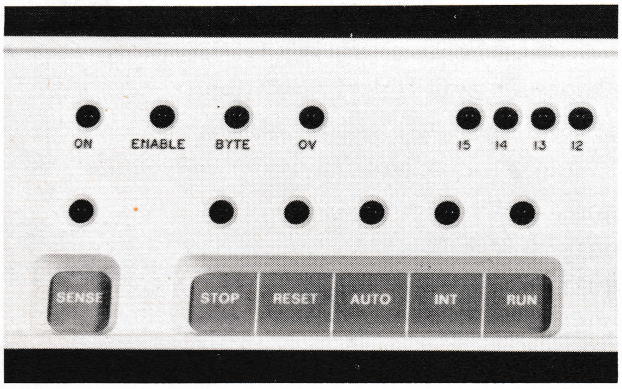


Fig 3.15 Start- och stopp-knapparna.

	STOP	RUN	
stoppa	tänd	släckt	STOP MODE
stega	tänd	tänd	STEP MODE
klart för start	släckt	släckt	RUN ENABLE MODE
starta	släckt	tänd	RUN MODE

Tänder man STOP-lampan stannar datorn. Med RUN-knappen kan man därvid stega fram datorn med en instruktion i taget. När instruktionen utförts återgår datorn i STOP MODE.

Med både STOP- och RUN-lamporna släckta är datorn klar för start och detta kallas RUN ENABLE MODE. Därvid är de funktioner som vi utnyttjat för in- och avläsning av registren (siffertangenter, display och registertangenter) urkopplade.

Datorn startas från RUN ENABLE MODE genom att man trycker på RUN-knappen.

INT betyder interrupt, dvs avbrott. INT-knappen genererar en speciell avbrottssignal från kontrollpanelen.

AUTO används vid automatisk laddning av program från TTY och andra I/O-enheter.

SENSE-lampan längst till vänster kan avkännas under programkontroll. SENSE-, AUTO- och INT-lamporna skall vara släckta vid våra försök.

Vi har nu bekantat oss med kontrollpanelen och vi har också skrivit ett program och lagrat det i minnet. Vi är därmed mogna för det första försöket.

3.1.5.5 Körningen

Man talar ofta om att "köra" eller "exekvera" ett program. Vi ska nu köra programmet i fig 3.13. Kontrollera först att programmet ligger kvar i minnescellerna 0 - 3 och att de tal som skall adderas är inskrivna i cell 4 - 5. Nollställ sedan programräknaren på det sätt vi beskrev på föregående sida.

Ställ datorn i RUN ENABLE MODE, dvs med både STOP- och RUN-lamporna släckta. Starta körningen med RUN-knappen.

Vårt program körs på några få mikrosekunder och det är alltså klart långt innan vi hinner ta fingret från RUN-knappen. Datorn står nu i RUN ENABLE MODE men har stannat under programkontroll eftersom vi har lagt in en stoppinstruktion (HLT) efter slutförd addition

Vill vi nu avläsa programräknarens ställning och resultatet, måste vi först ställa datorn i STOP MODE (Tryck på STOP-knappen).

Ett tryck på P-knappen ger resultatet 0004 i indikatorn. Datorn har alltså vid HLT-instruktionen räknat fram programräknaren ett steg och sedan stannat.

Resultatet av vår addering ska nu vara lagrat i minnesposition 6. Låt oss kolla innehål-

let i cellerna 4, 5 och 6 genom att först ställa programräknaren på 0004 och sedan trycka på M-knappen. Vi får successivt

M	0003
M	0004
M	0007

Datorn har tydligen utfört den önskade additionen: 3 + 4 = 7.

Pröva med några fler exempel men kom ihåg att datorn räknar binärt. Vad blir summan av 0ABC och 0DEF exempelvis? (Blir summan 18AB? Räkna ut summan både för hand och med hjälp av ALPHA LSI!)

3.1.6 VAD HAR VI LÄRT OSS?

Vi har nu gjort den första handgripliga bekantskapen med datorn ALPHA LSI. Frånsett knapparnas placering, märkning och manövering fungerar de flesta minidatorer på likartat sätt.

Vi har sett att vår dator har stora likheter med den tidigare beskrivna kalkylatorn, både när det gäller blockschema och programmering.

För att skriva ett program till en minidator måste vi vara väl insatta i datorns instruktionslista. Kalkylatorprogrammet var däremot enklare att göra eftersom instruktionerna hängde så intimt samman med tangenternas normala användning vid manuell beräkning med kalkylatorn.

I datorn måste vi hålla reda på var programmet ligger lagrat och ställa in programräknaren på rätt startadress. I kalkylatorn hade vi ett separat programminne och därför uppstod aldrig problem med programmets placering.

Datorn är inte avsedd att köras från kontrollpanelen och därför har vi haft litet extra-arbete med alla knapparna. Men dessa knappar ger oss å andra sidan generella möjligheter att ingripa i och kontrollera datorns arbete.

När man ger sig i kast med en minidator blir en stor del av arbetet att läsa manualer. De ger den detaljerade och exakta beskrivning av datorn som är ofrånkomlig för det fortsatta arbetet - men som samtidigt gör manualerna till en svårläst och svårsmält faktasamling. Fig 3.16.

En av avsikterna med detta häfte är bl a att ge en viss grundförståelse för vad som gömmer sig bakom manualernas kortfattade rubriker.

Vi har exempelvis i detta avsnitt berört instruktionslistan som upptar stort utrymme i "Programning reference manual". Vi har också bekantat oss med kontrollpanelen (console) som finns noggrant beskriven i programmeringsmanualen.

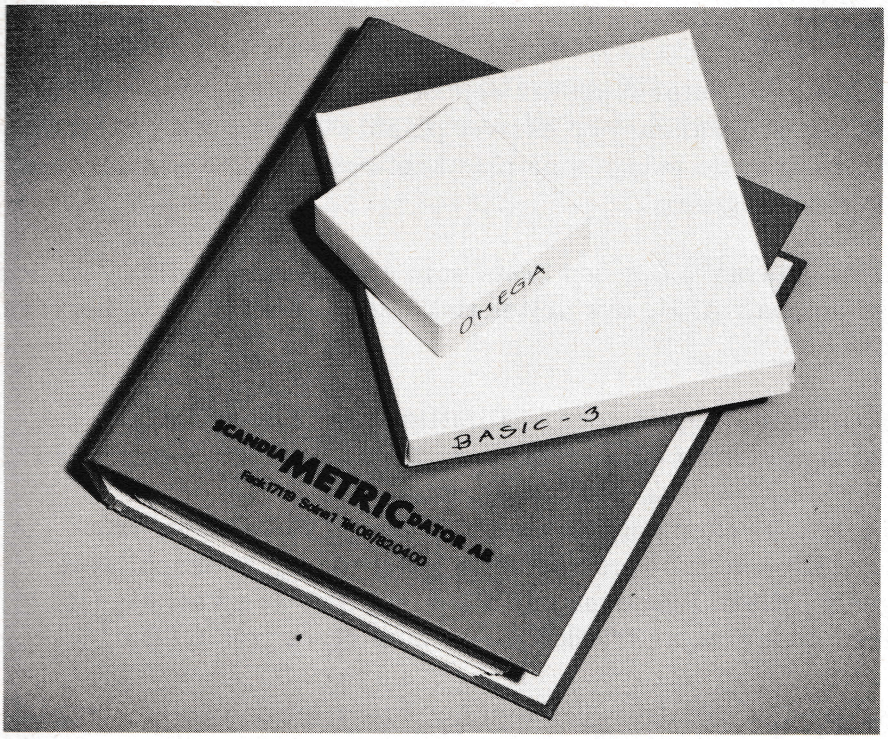


Fig 3.16 Pärm innehållande ALPHA LSI:s mamualer samt kartonger innehållande programvaror i form av hålremsor.

3.2 Något om elektroniken

I föregående avsnitt har vi sett att vår dator fungerar och kan utföra ett program som vi sätter i dess minne - förutsatt att programmet är skrivet på ett sätt som datorn kan tolka. Det är nu dags att lyfta på slöjan över de många små plastkretsarna på CPU-kortet, så att vi får en uppfattning om hur de fungerar.

Vi ska nedan börja med CPU:ns "hjärna", dvs den krets som utför de aritmetiska och logiska operationerna . Den kallas normalt ALU, dvs aritmetisk logisk enhet. Vi ska begränsa oss till dess funktion vid addition.

De tal vi bearbetar lagras på olika ställen i datorn. Tillfälliga lagringsplatser är latchar och register. För lagring av stora datamängder används minnen.

Talen, dvs data, måste ju förflyttas, exempelvis från I/O-enheten till minnet, från minnet till ALU:n osv. I datorn sker detta längs speciella busslinjer och inkopplingen till dessa linjer sker via signalgrindar. Bussarna och signalvägarna är så att säga datorns vägsystem där trafiken dirigeras av trafiksignaler som styrs från CPU:n.

Själva hjärtat i datorn är styrenheten (control unit) som ger order om vilka signalvägar som skall öppnas och i vilken takt data skall släppas fram i systemet. Som "pace maker" fungerar därvid en klocka, vanligen i form av en kristalloscillator.

Viljan eller "intelligensen" i systemet ligger i det program som vi placerat i minnet. Det är från detta program som styrenheten hämtar sina instruktioner. Och som vi redan sett är styrenheten mycket formell av sig och tolererar inga avsteg från de i instruktionslistan preciserade instruktionerna.

3.2.1 RÄKNEENHETEN

Både i dosan och kalkylatorn utförde vi en addition av två tal. Även datorn kan, som vi sett programmeras att utföra en addition. Hur gör man då en addition elektroniskt ?

3.2.1.1 Adderaren

Låt oss utföra additionen av de två binära talen A och B i fig 3.17.

1 1 1 1	överföringssiffra (carry)
0 1 0 1 1 1 0 0	talet A (= 92)
+0 0 1 1 1 0 1 0	talet B (= 58)
1 0 0 1 0 1 1 0	summan S (= 150)

Fig 3.17 Binär addition

Vid additionen börjar vi som vanligt med siffran längst till höger (LSB = least significant bit). De två första positionerna (räknat från höger) kan vi addera utan att behöva ta hänsyn till någon överföringssiffra (minnessiffra, carry). Här kan vi använda en enkel regel för addering av två binära siffror. Den kan formuleras som en tabell enligt fig 3.18a.

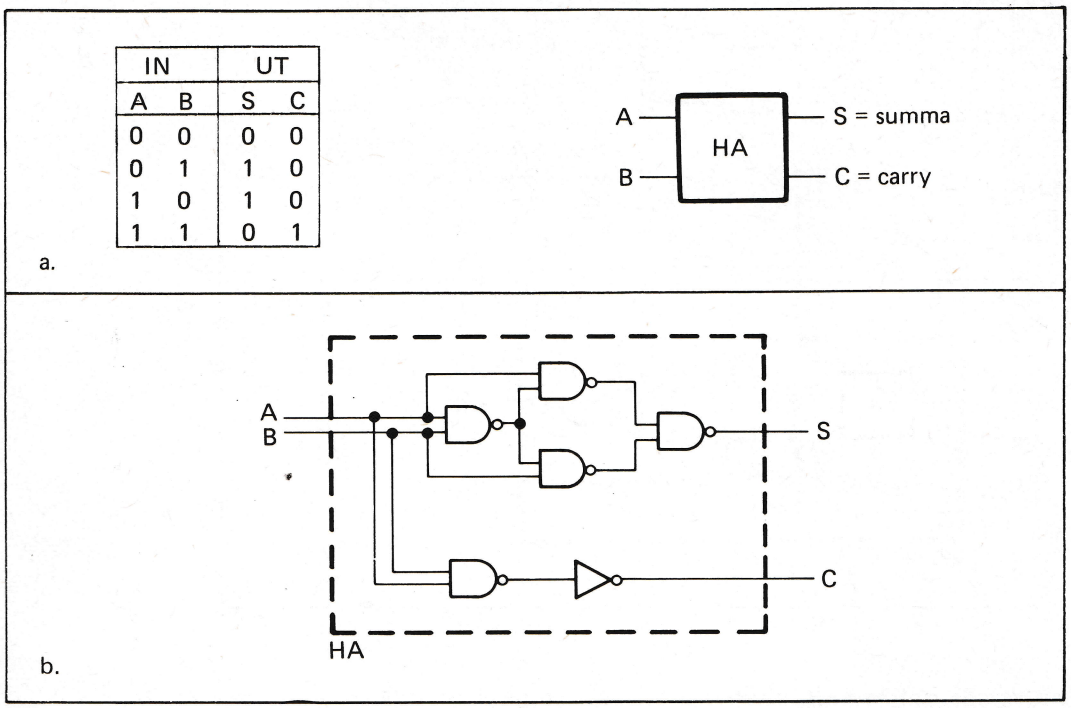


Fig 3.18 Halvadderaren (HA)
a. Räkneregel och blockschema
b. Halvadderare uppbyggd av enkla grindar

Eftersom det är ett entydigt samband mellan instorheterna (A, B) och utstorheterna (S, C) kan vi bygga upp en elektronisk krets av enbart enkla grindar - som ger just detta samband.

En sådan krets kallas halvadderare (HA = half adder) och kan vara uppbyggd på det sätt som visas i fig 3.18b.

Vid additionen av nästa position (den tredje från höger) i fig 3.17 måste vi ta hänsyn till en överföringssiffra. Vi måste då använda en regel för addition av tre siffror och den kan formuleras som i fig 3.19a. En krets som lyder denna regel kan se ut som i fig 3.19b. Eftersom den kan ta hand om en carry på ingången kallas kretsen för heladderare (FA = full adder).

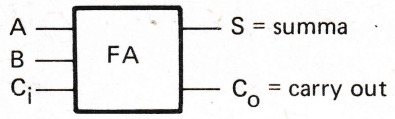
Med hjälp av en heladderare kan vi addera två binära tal rent elektroniskt. Fig 3.20 visar principen.

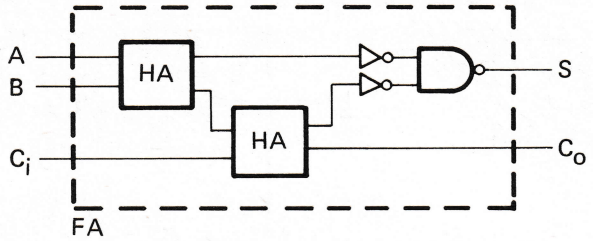
Talen (A och B) placeras i var sitt register och skiftas - en bit i taget in i heladderaren. Summan S skiftas ut i registret S på utgången.

Eftersom vi vid varje sifferaddition måste spara carrysiffran ("i minnet") till efterföljande sifferaddition, så har vi satt in en vippa mellan carryutgången C_0 och carryingången

IN			UT	
A	B	C_i	S	C_o
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

a.





b.

Fig 3.19 Heladderaren (FA)
a. Räkneregel och symbol
b. Uppbyggnad

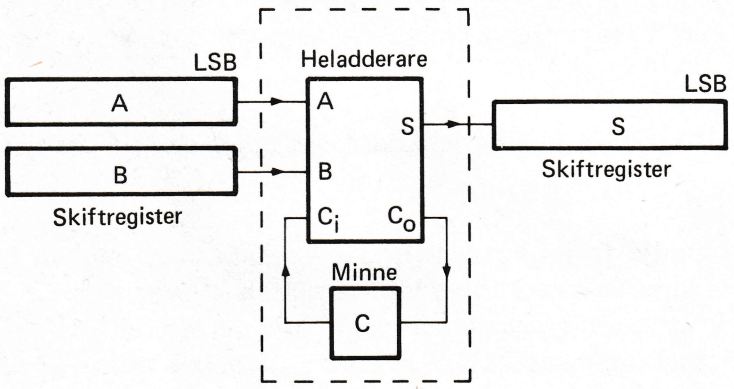


Fig 3.20 Serieaddition.

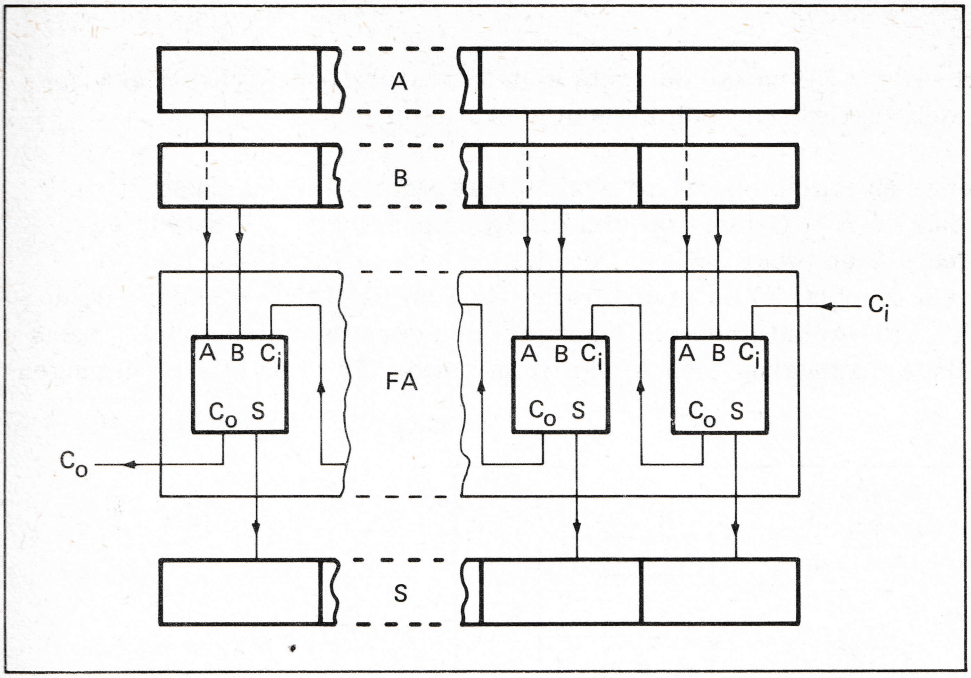


Fig 3.21 Parallelladdition.

C_i . Vippan har en speciell utformning. Den innehåller som de flesta IC-vippor egentligen två vippor, en mästare (M) och en slav (S), och kallas därför MS-vippa. MS-vippan har förmågan att vid skiftpuls ta emot ett värde på ingången (dvs C_o) och samtidigt hålla kvar det föregående värdet på utgången (dvs C_i). x)

Additionen i fig 3.20 görs i serieform - dvs precis på det sätt som när vi adderar för hand. Eftersom siffrorna stegvis måste skiftas igenom adderaren tar det tid. Å andra sidan behövs enbart en adderare. Så kretsen blir billig.

Serieaddering används både i räknedosan och kalkylatorn, för här är ju tiden ingen begränsning.

En dator skall arbeta snabbt och då måste man koppla ett antal adderare parallellt så som visas i fig 3.4. Adderingen sker snabbt - men carry måste förstås fortplanta sig genom hela kedjan innan resultatet på utgången blir rätt. Addition av denna typ kallas parallelladdition och det är denna typ av addition som sker i minidatorn.

Vi har tidigare sett att en subtraktion kan överföras till en addition. Istället för att subtrahera ett tal så tar man och adderar med talets 2-komplement. Därmed kan vi använda adderaren även för subtraktion.

---

x) MS-vippan beskrivs i Markesjö, Elektronik, Inledande kurs B, Läromedelsförlagen, Stockholm 1970.

3.2.1.2 ALU:n

Aritmetisk logisk enhet eller ALU kallas en krets som kan utföra både aritmetiska operationer (som addition och subtraktion) och logiska operationer.

Man frågar sig kanske hur en logisk operation skall tolkas i detta sammanhang. Den enkla grinden i fig 1.4 har en NAND-funktion (om vi låter spänningen 5 V betyda en binär etta och spänningen 0 V en binär nolla). Det är med sådana NAND-kretsar vi ovan byggt upp adderaren. Kopplar vi en inverterare (fig 1.8) på NAND-grindens utgång får vi en AND-grind. En AND-grind (med två ingångar) och dess "sanningstabell" visas i fig 3.22. Enbart när båda ingångarna (dvs A och B) har ettor får vi en etta på utgången - därav namnet.

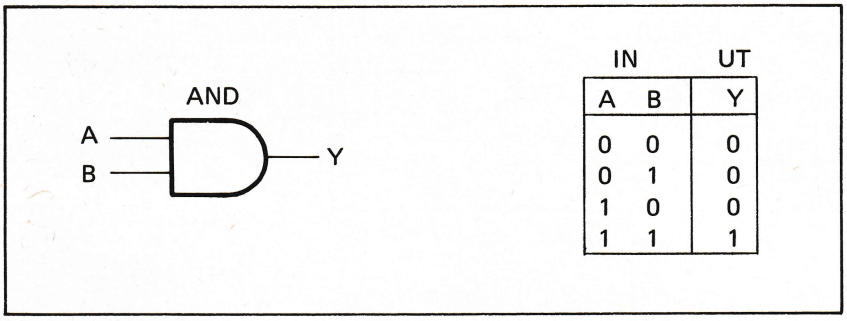


Fig 3.22 AND-funktionen.

Antag nu att vi har två register, vardera med 16 bitar. Vi kan då låta bitarna i dessa register parvis mata AND-grindar så som visas i fig 3.23 och placera resultatet i ett tredje resultatsregister R. Man kan jämföra funktionen med signalgrinden i fig 1.7 och tolka resultatet i R-registret som A-registrets innehåll styrt (eller "avmaskat") genom B-registret (eller vice versa).

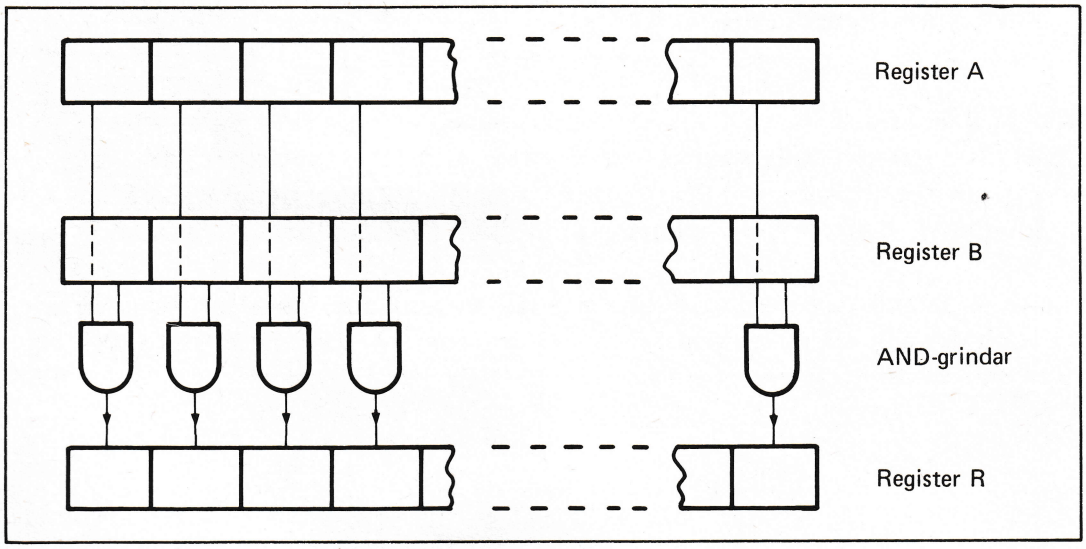


Fig 3.23 "AND-ing" av två register.

Ett praktiskt exempel förklarar kanske enklare vad "AND-ing" eller "MASK-ing" innebär. Antag att vi enbart vill släppa fram de 8 minst signifikanta bitarna i ett 16-bitars ord som just är lagrat i B-registret. Vi behöver då bara sätta en "mask" med följande utseende i A-registret.

	15							8	7							0
A-register	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
	maskerar position 8-15								släpper fram position 0 - 7							

Med AND-funktionen får vi bitarna 0 - 7 från B-registret direkt överförda till R-registret medan bitarna 8 - 15 spärras.

Det hade givetvis gått lika bra att använda NAND-kretsar istället för AND-kretsar. Enda skillnaden hade då blivit att vi fått de till R-registret framsläppta bitarna 0 - 7 invererade.

Aritmetiska och logiska funktioner är alltså mycket användbara vid hantering och bearbetning av data.

En ALU består enbart av grindar. Lägger man på en signalkombination på ingångarna

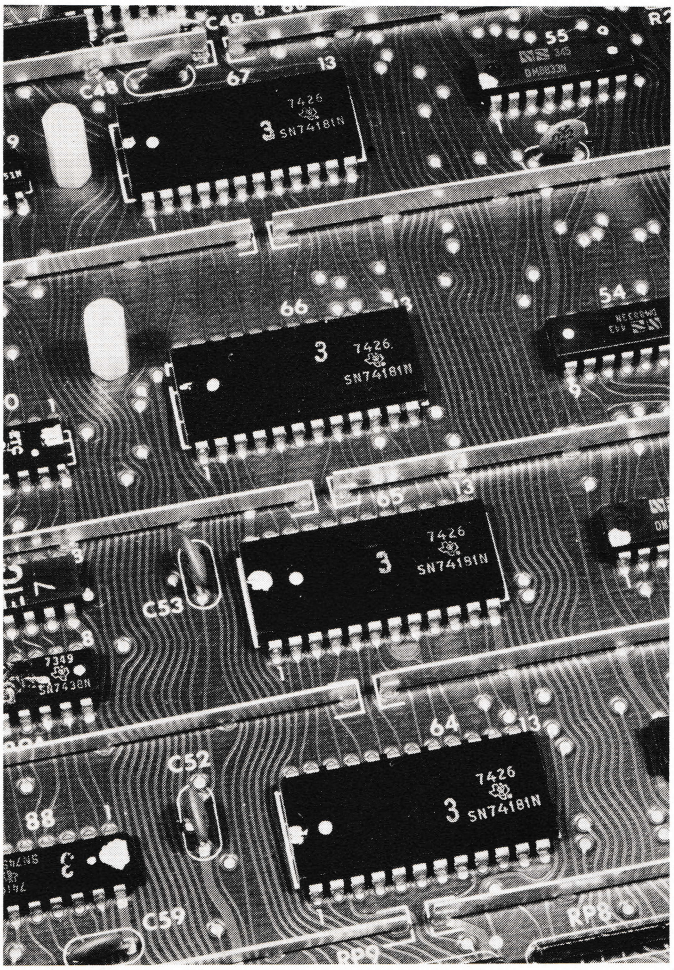


Fig 3.24 ALU:n i vår dator gömmer sig i fyra kapsler.

får man därför en entydig kombination av spänningar på dess utgångar. Dessa utsignaler representerar därvid exempelvis en summa, en skillnad, AND-ing eller andra aritmetiska eller logiska resultat av de pålagda insignalerna. Vilken funktion som ALU:n utför beror på inställningen av dess styringångar.

Fig 3.24 visar de fyra ALU-kretsar som tillsammans hanterar två 16-bitars tal i ALPHA LSI. Här sitter alltså "hjärnan" som räknar och utför de aritmetiska och logiska operationerna i vår dator.

ALU:n lagrar ingen information. Presenterar vi två 16-bitars tal på dess ingångar svarar den med en funktion av dessa tal på utgångarna, men så fort vi tar bort insignalerna försvinner utsignalerna.

Fig 3.25 visar innehållet i en av kretsarna. Den består av ca 75 grindar, integrerade i en och samma kristall. Kretsen heter 74181 och kan utföra inte mindre än 32 olika funktioner (16 aritmetiska och 16 logiska), bl a följande:

aritmetiska funktioner:

addition
subtraktion
skift ett steg (dvs multiplikation eller division med 2)
m fl

logiska funktioner:

AND
NAND
jämföra (comparator)
m fl

3.2.2 ATT LAGRA DATA

Vi har tidigare sett (fig 1.4) att en vippa kan lagra en binär siffra, dvs en bit. På ett flertal ställen i vår dator har vi behov av att på olika sätt lagra data. Det enklaste sättet att lagra är med hjälp av fristående vippor som ofta kallas latchar.

Med register menar man ofta vippor som på något sätt är hopkopplade, exempelvis så att en bit kan skiftas från vippa till vippa i registret.

Med minne avser man som regel kretsar som kan lagra många bitar och som exempelvis innehåller ett stort antal vippor. Man blir inkopplad till en eller flera av dessa vippor genom att tillföra en adress - dvs en spänningskombination - på minnets adressingångar.

Vi ska nu se exempel på några kretsar för datalagring på vårt CPU-kort.

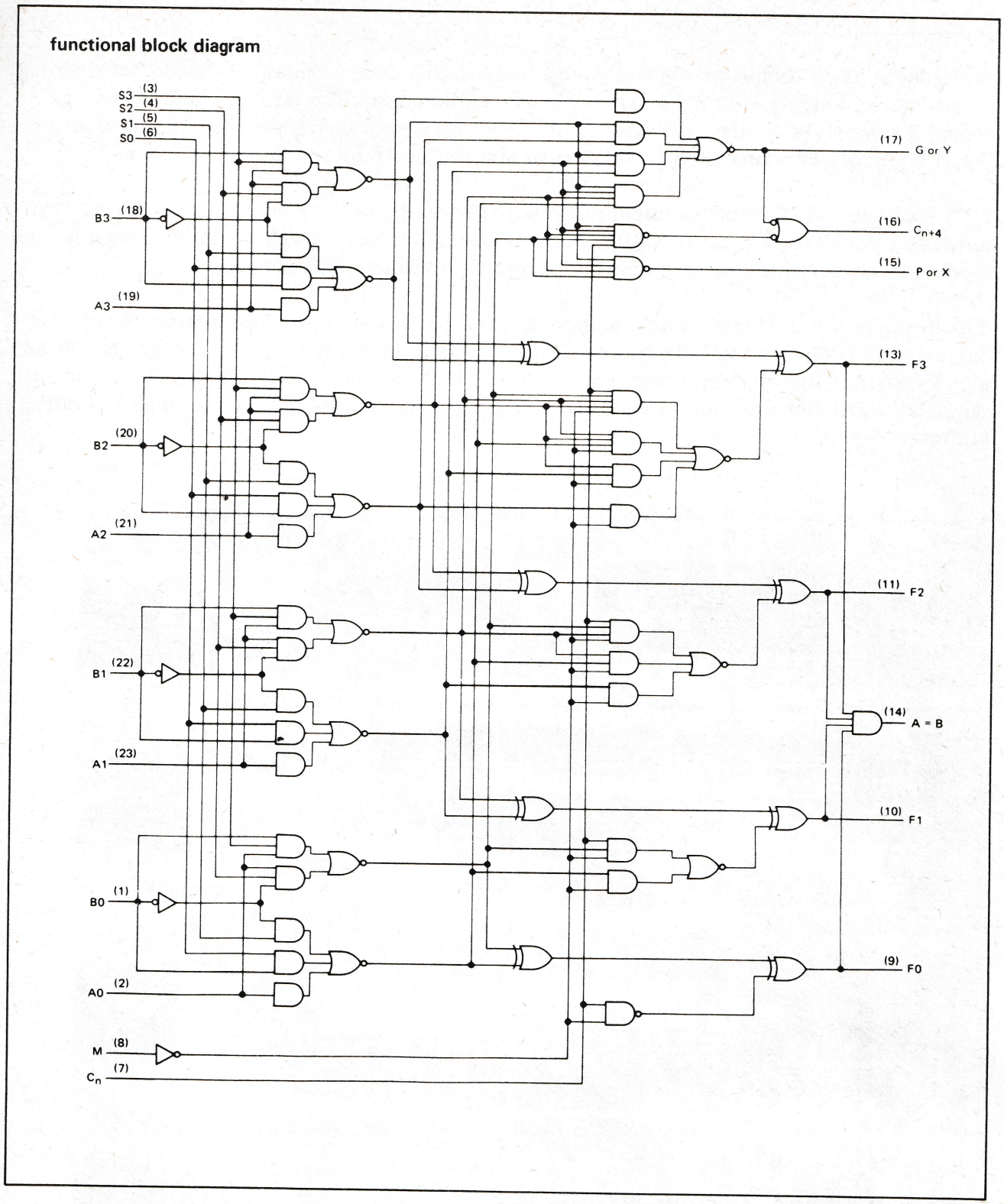


Fig 3. 25 Innehållet i ALU:n 74181 (Texas Instruments).
Utöver de i texten beskrivna kretsarna (AND, NAND och inverterare) innehåller figuren även NOR- och EXOR-kretsar. För en beskrivning av dessa hänvisas till litteraturreferens 4 i slutet av boken.

3.2.2.1 En latch

I vår dator har vi behov av att kortvarigt kunna lagra data i väntan på vidare bearbetning. Denna typ av lagring sker i s k latchar (eller hållkretsar). En latch är helt enkelt en vippa, exempelvis av den typ vi sett i fig 1.4a. Eftersom vi räknar med 16 bitars ordlängd åtgår det 16 vippor och då talar man ofta om en 16 bitars latch.

TTL-kretsen 74174 är ett exempel på en kapsel som innehåller en sexbitars latch. Den används i vår dator för att tillfälligt lagra (eller hålla) det till ackumulatorn inkommande talet, medan beräkningen med ackumulatorns tidigare innehåll slutförs.

Till höger på fig 3.26 ser vi ackumulatorlatchen på CPU-kortet. Den består av tre kretsar av typ 74174 (dvs totalt 18 bitar). Fig 3.27 visar innehållet i en av kretsarna. Vi ser att varje vippa har en dataingång och en datautgång. Inläsning sker då CLOCK-ingången läggs hög. Det finns dessutom en CLEAR-ingång med vars hjälp alla vipporna kan nollställas.

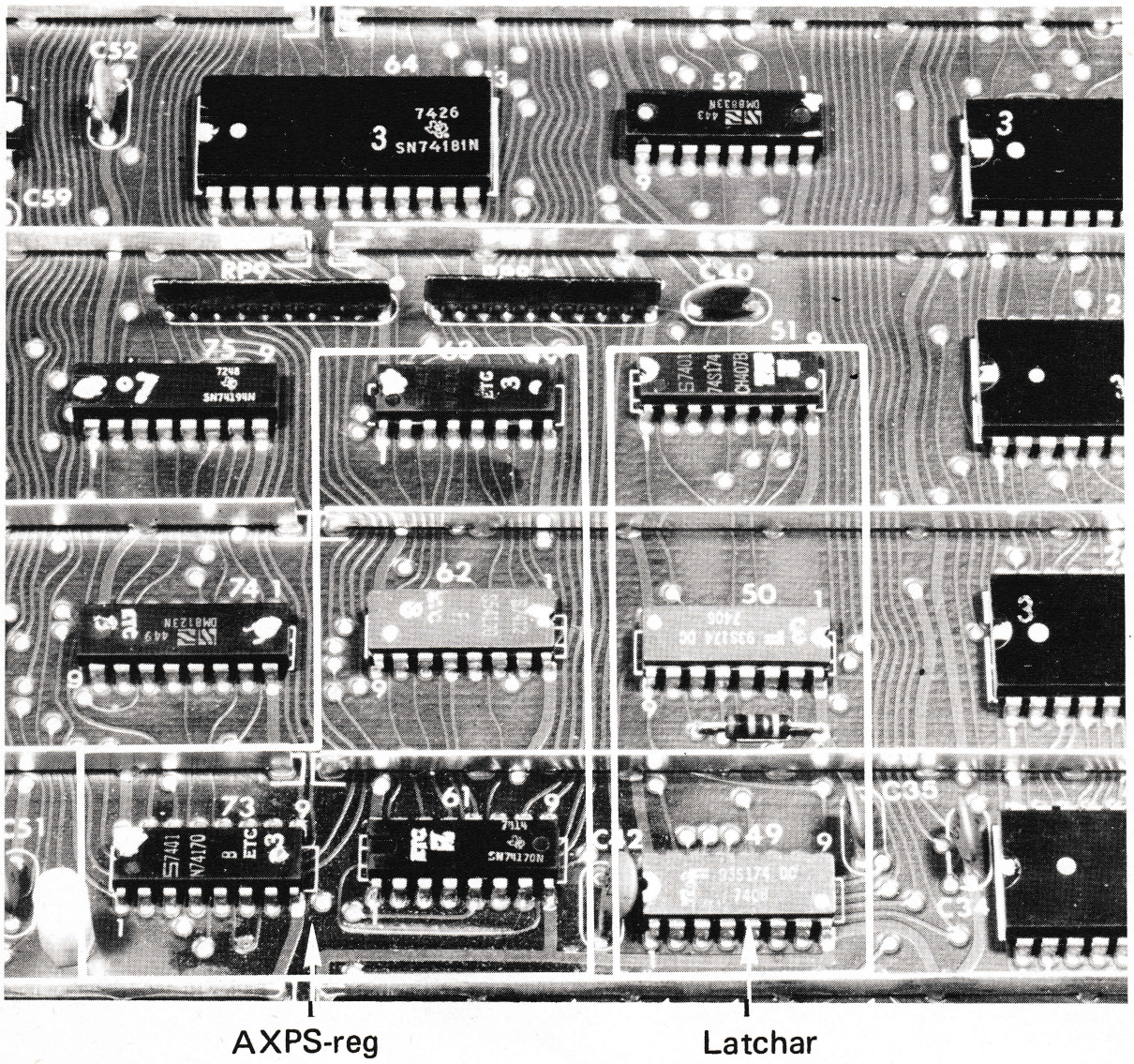


Fig 3.26 Detaljbild från CPU-kortet. Till vänster fyra sextonbitars register (A,X,P och S) av typ 74170. Till höger tre sexbitars latchar (74174).

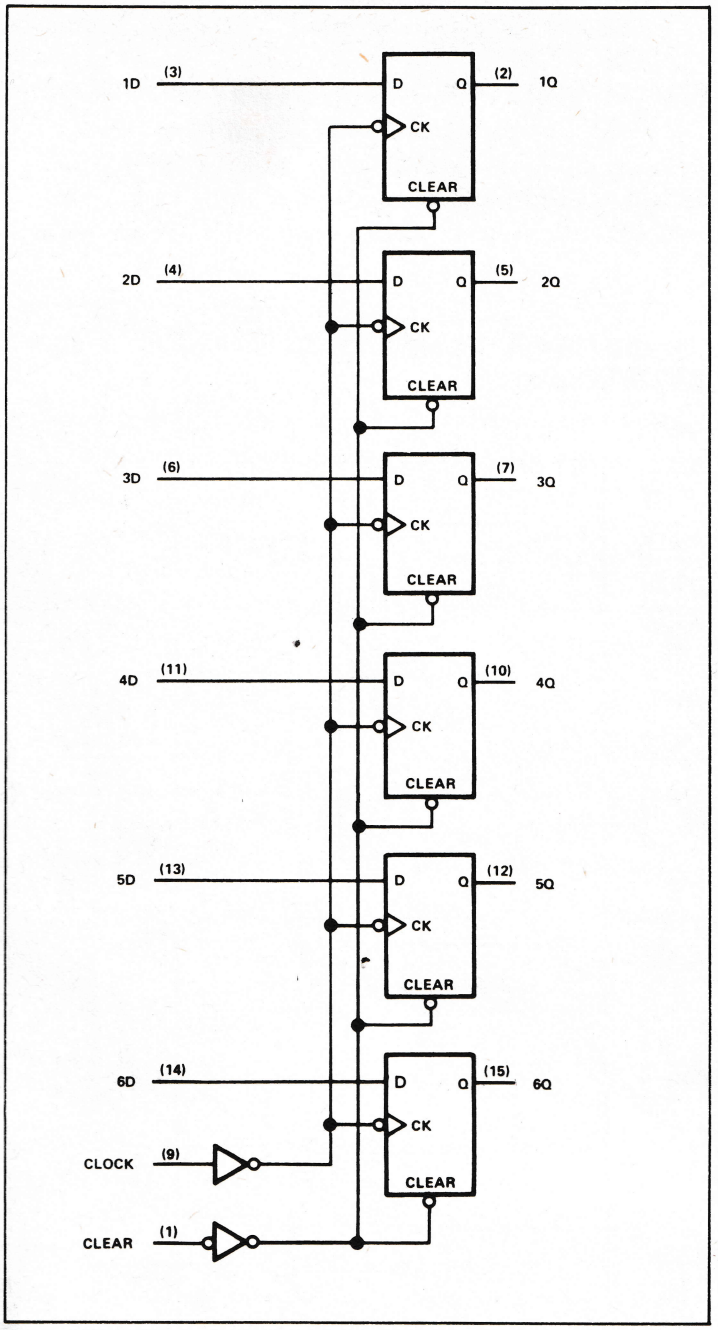


Fig 3.27 Innehållet i en sexbitars latch (Texas Instruments, 74174).

3.2.2.2 Ett register

I vår dator används flera typer av register. Fig 3.28 visar de fyra kapslar som innehåller M- och W-registren på CPU-kortet.

Varje kapsel innehåller 8 vippor och dessutom ett grindnät som gör det möjligt att läsa in- eller ut data i parallell- eller serieform och att stegvis skifta data till vänster eller till höger inom registret.

Kretsarna heter 74198 och innehållet i en sådan kapsel framgår av fig 3.29. Det är som vi ser ganska innehållsrika plastkapslar på CPU-kortet!

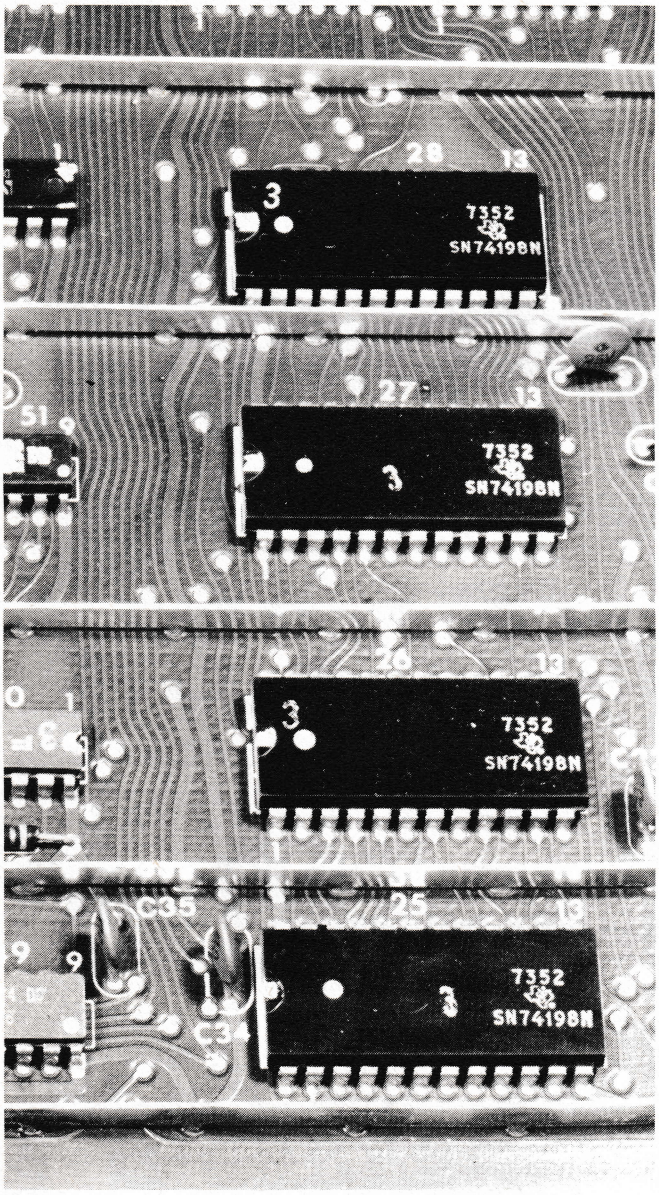


Fig 3.28 M- och W-registren på CPU-kortet.

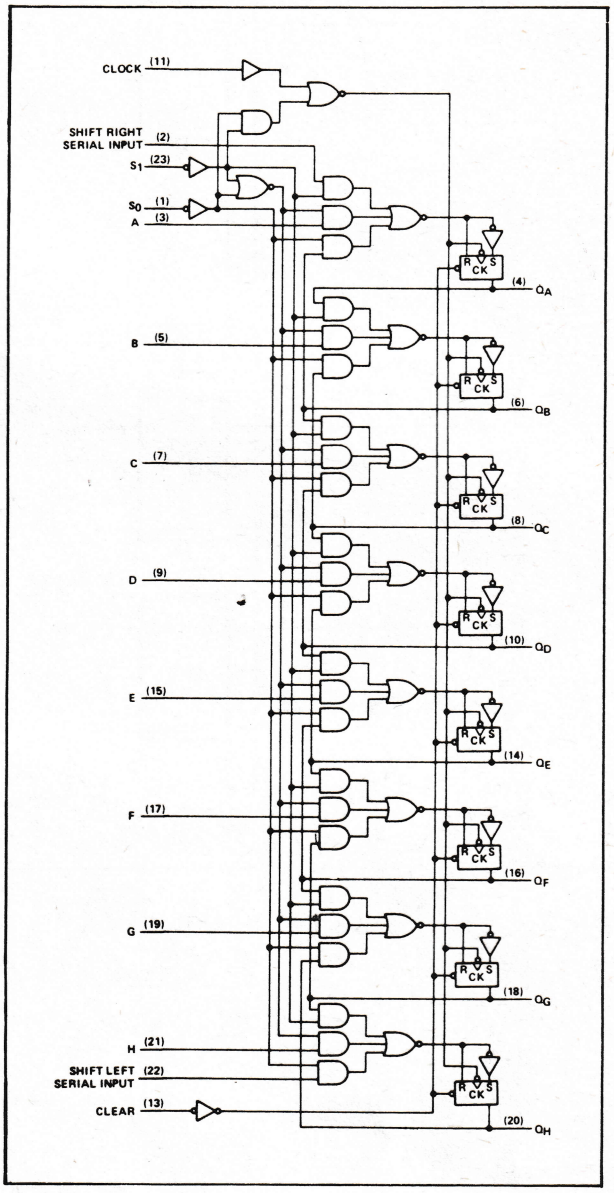


Fig 3.29 Innehållet i kretsen 74198.

3.2.2.3 Olika typer av minnen

Typiskt för ett elektroniskt minne är att det innehåller data som är lagrade i fack (eller celler). Varje fack har sin egen adress. För att ge minnet adressen till det sökta facket så lägger man på en kombination av spänningar på adressingångarna. Eftersom det är binära värden, åtgår det N st adressingångar för att adressera 2^N st ord i ett minne.

3.2.2.3.1 Ett ROM

Fig 3.30 visar principen för ett läsminne. Det kallas ofta för ROM (read only memory) eller ROS (read only storage). Ett ROM har permanent lagrad information och kan därför enbart läsas. Lägger vi på en adress på ingången, få vi ut innehållet i det adresserade facket på utgången.

Eftersom ROM:et i fig 3.30 har 8 adressingångar, kan det adressera $2^8 = 256$ ord. Minnet har fyra datautgångar och data kommer ut som ord om vardera fyra bitar på utgången. Man säger att ett sådant ROM är organiserat i 256 ord om 4 bitar och dess kapacitet är 256 x 4 = 1024 bitar, dvs 1 kbit.

I vår dator är instruktionerna lagrade i ett speciellt mikroprogramminne som består av 14 kretsar av den beskrivna typen. Vi ser mikroprogramminnet på det "halvkort" som sitter monterat på det stora CPU-kortet, fig 3.31. Totalt innehåller dessa 14 kapslar 256 ord om vardera 56 bitar (dvs 14 kbit).

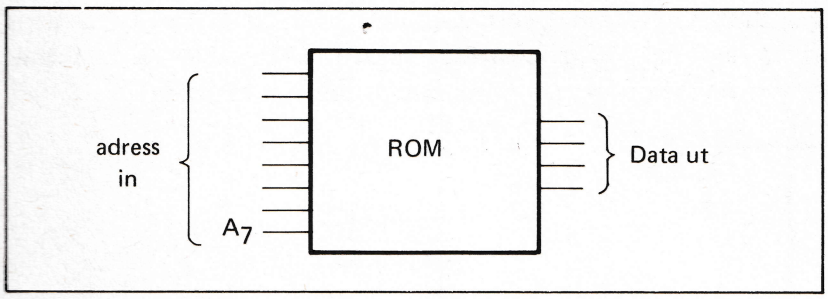


Fig 3.30 Ett ROM (256 ord om 4 bitar).

Eftersom ett ROM har en enkel uppbyggnad och dessutom är en vanlig krets i digitala system, ska vi studera det litet mer i detalj. Men vi tar ett mindre ROM för vår detaljstudie, ett som innehåller enbart 8 ord om vardera fyra bitar.

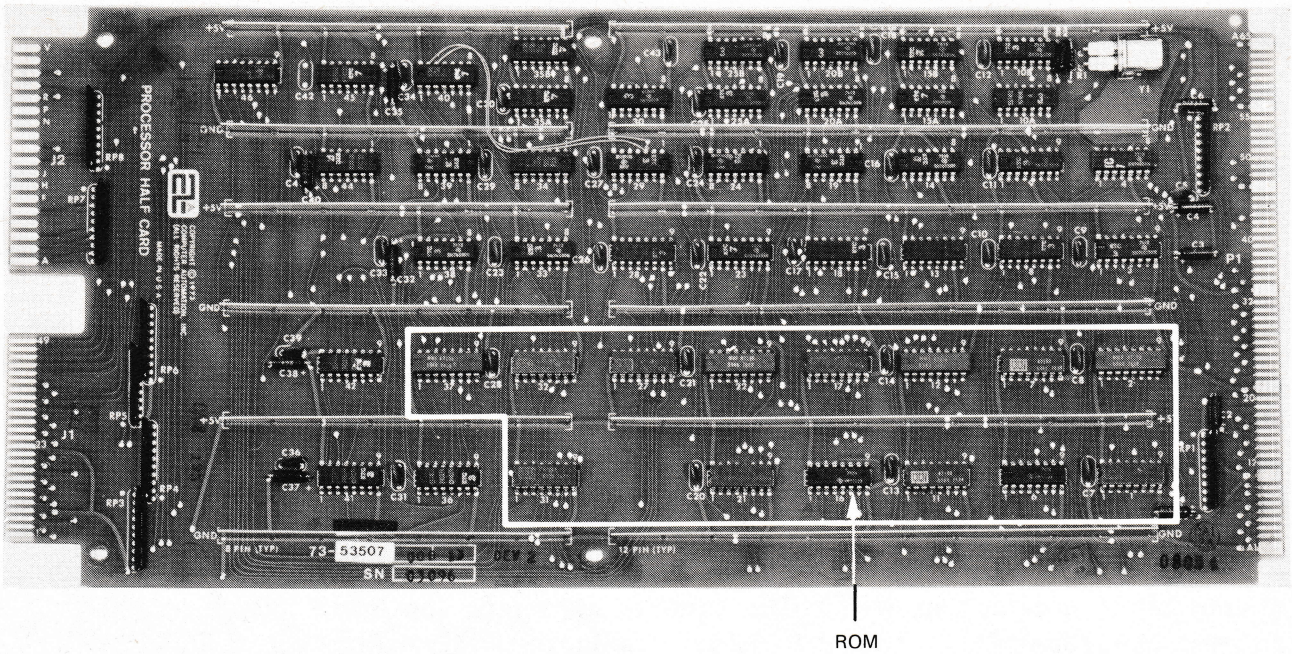


Fig 3.31 ROM som lagrar mikroprogrammet i ALPHA LSI.

För att välja ut ett av de åtta orden i minnet åtgår 3 adressledningar ($2^3 = 8$). I fig 3.32 har vi ritat en adressavkodare som har åtta utgångar - en till varje ord i minnet. För varje adress, dvs för varje kombination av inspänningar är enbart en av dessa utgångar (ordledningar) låg. Vi kallar kretsen i fig 3.30 för adressavkodare och den är uppbyggd av just sådana enkla NAND-grindar som vi tidigare sett exempel på i fig 1.4b. I vissa av NAND-grindarnas ingångar har inverterare (fig 1.8) inkopplats. För att göra schemat överskådligare har dessa inverterare enbart utritats som ringar på ingångarna.

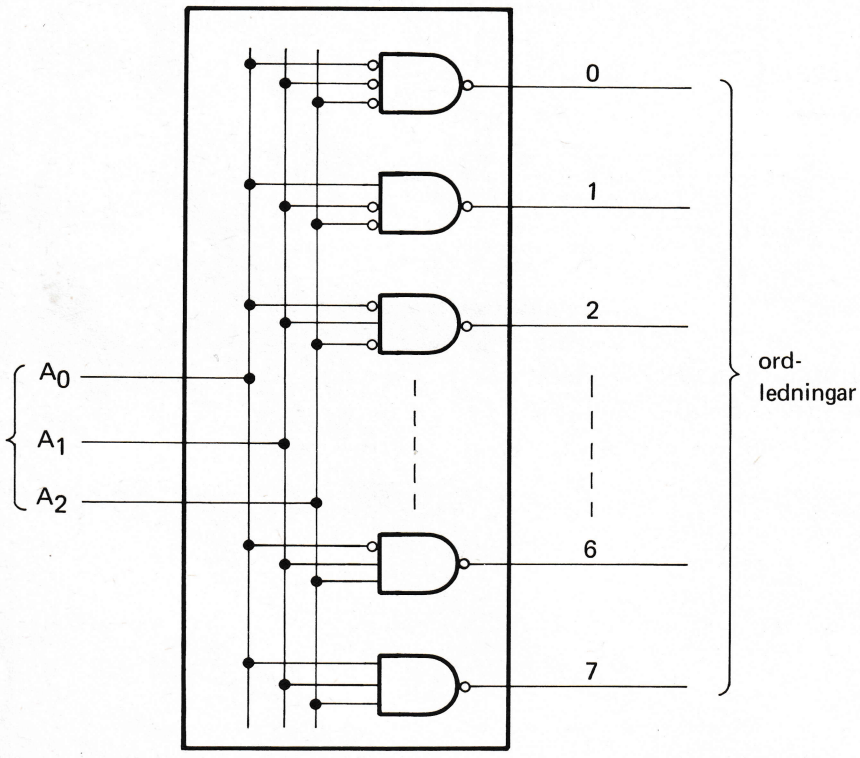


Fig 3.32 En adressavkodare ("en av åtta").

Nu har vi alltså avkodat den inkommande adressen och lagt en av de åtta ordledningarna låg (dvs på spänningen 0 V).

I fig 3.33a ser vi uppbyggnaden av själva minnesmatrisen. Den består av våra tidigare beskrivna åtta ordledningar och därtill fyra överkorsande bitledningar. Det är givetvis fråga om en "tvåplanskorsning" och alla ledningar är isolerade från varandra.

Observera att samtliga bitledningar ligger höga, dvs på spänningen 5 V. De matas ju via serieresistorer (1 kΩ) från matningsspänningen (H = + 5 V).

Vi tänker oss nu att den binära adressen 111 (=7) påläggs adressingången. Samtliga adressledningar (A_0, A_1 och A_2) ligger alltså höga. Resultatet blir att samtliga ordledningar utom nr 7 ligger höga. Ordledning 7 ligger däremot låg.

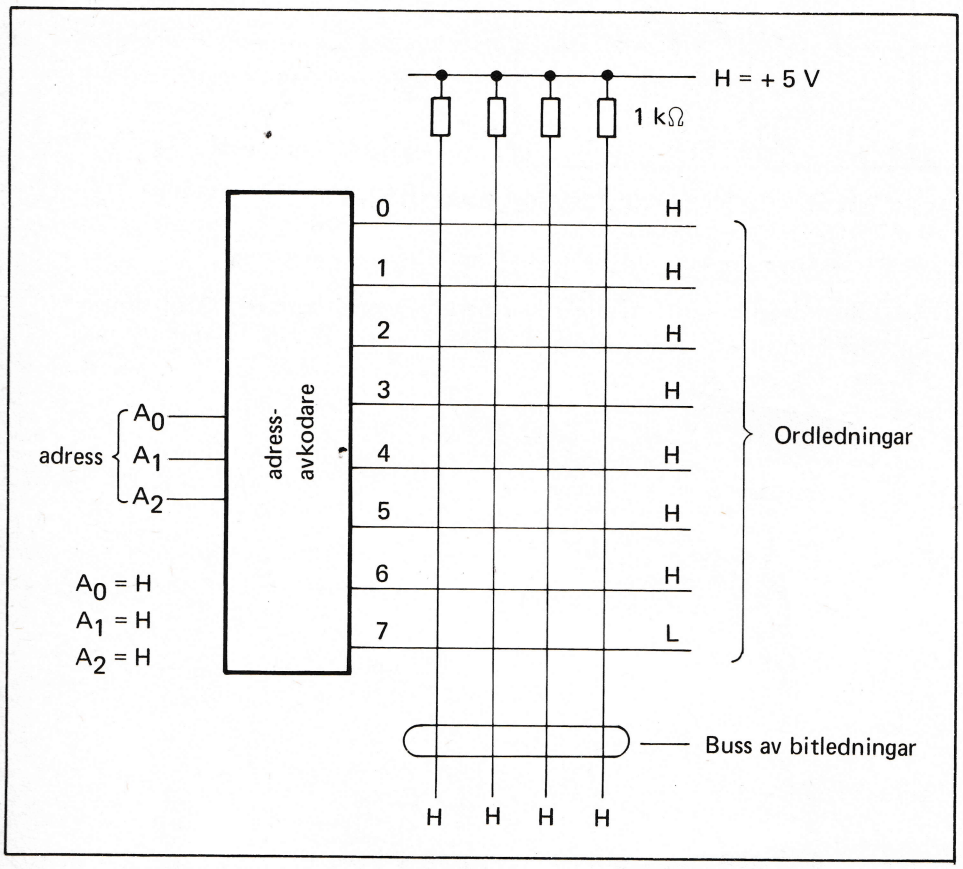


Fig 3.33a. Uppbyggnaden av ett ROM: Ordledningar och bitledningar.

	IN adress	UT ord	Matris
0	000	1101	
1	001	0101	
2	010	1001	
3	011	1111	
4	100	0000	
5	101	0000	
6	110	1101	
7	111	0000	

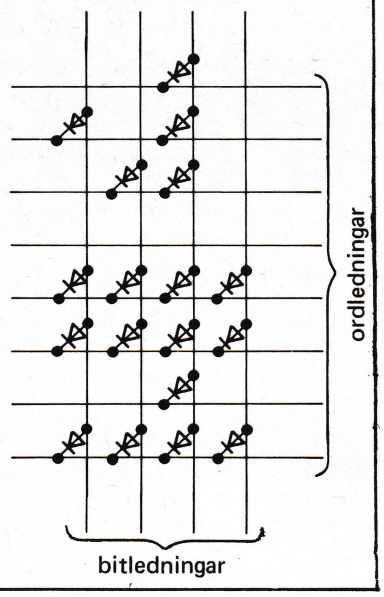


Fig 3.33b. Uppbyggnaden av ett ROM: Programmering med dioder.

Nu har vi lärt känna adressavkodningen och minnesmatrisens uppbyggnad. Men ännu ligger ingen information lagrad i minnet! Nu kommer den!

Låt oss lagra orden

1101
0101
1001
1111
0000
0000
1101
0000

på adresserna 0 till och med 7!

I fig 3.33b har vi formulerat våra önskemål i tabellform och själva datalagringen - eller programmeringen som man brukar kalla det - sker nu med hjälp av en lödkolv och ett antal dioder.

Låt oss börja bakifrån med ord nr 7 (dvs med adressen 111). Det skall enligt tabellen vara 0000 vilket betyder låg spänning på samtliga bitledningar. Kopplar vi nu in fyra dioder från ordledningen 7 (som ju ligger låg vid adressen 111), en till varje bitledning så som visas i fig 3.33b, så sänker vi spänningen på samtliga utgångar.

Varje nolla kostar oss på detta sätt en diod. Ord nr 6 innehåller endast en nolla och den motsvaras alltså av dioden mellan motsvarande bitledning och ordledning 6.

Tack vare dioderna stör inte ordledningarna varandra. Det är bara den ordledning som ligger låg som har kontakt med bitledningarna via sina dioder.

Plastkapslarna i fig 3.31 innehåller vardera 1024 bitar och är givetvis inte tillverkade med hjälp av lödkolv och dioder. Här sker programmeringen på fotografisk väg i själva tillverkningsprocessen. Datorfabrikantens önskelista på innehållet i ett ROM sänds i form av en hålremsa till halvledarfabrikanten. Där omformas hålremsan (med en dators hjälp) till ett mönster som exponeras på kristallen och därvid bestämmer inkopplingen av dioder (I praktiken är det transistorer som inkopplas men det ändrar inte funktionen i princip).

3.2.2.3.2 Ett RAM

Ett ROM har, som vi sett ovan, permanent lagrad information. I vår dator behöver vi emellertid även skriv-läsminnen, dvs minnen som vi snabbt kan lagra ny information i (utan att ta fram lödkolven!)

Ett skriv-läsminne kallas vanligen RAM (random access memory). Ordet RAM antyder att man har lika snabb access till samtliga ord i minnet.

Ett RAM kan vara uppbyggt på i princip samma sätt som det ROM vi studerat i fig 3.31, men med en väsentlig skillnad. Minneselementen i ett RAM skall kunna programmeras med signaler och det kan vi göra om vi använder vippor istället för dioder i krysspunkterna. Vi ska inte ge oss in i detaljerna eftersom principen ju är ganska klar. För att kunna ställa vipporna i önskade lägen måste vi ha dubbelt så många bitledningar i minnesmatrisen, vi behöver ju lika många bitledningar för att skriva som för att läsa. Dessutom måste vi ha en kontrollingång som ger minnet order om det skall skriva eller läsa på den angivna adressen.

Principen för ett RAM framgår av fig 3.34.

En dators primärminne kan innehålla halvledarminnen av RAM-typ där varje kapsel inrymmer 4096 bitar (fig 3.6).

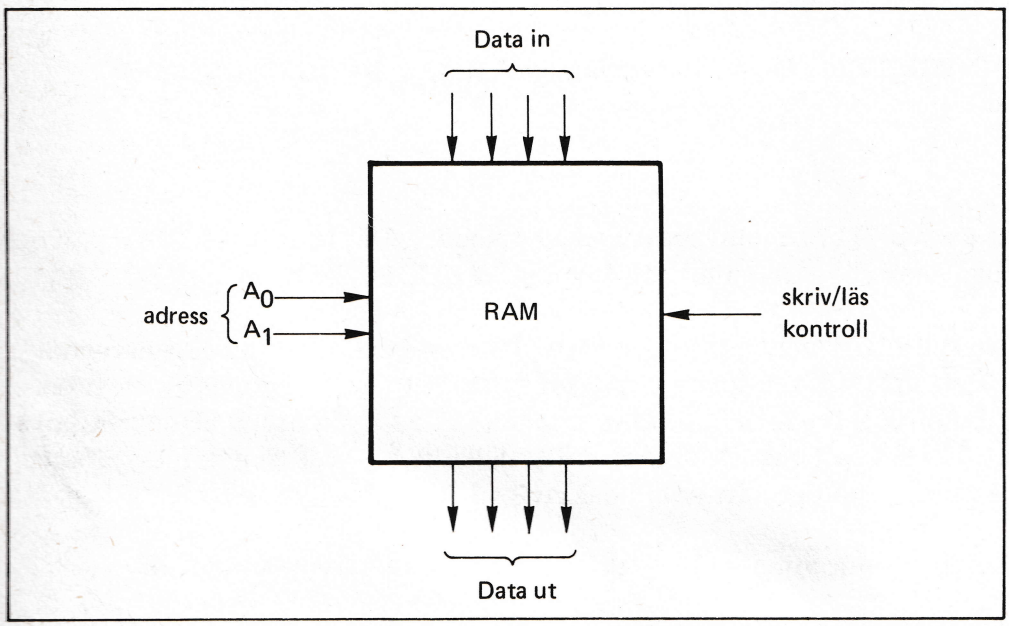


Fig 3.34 Principen för ett RAM.

3.2.2.3.3 En FILE (engelskt uttal) eller registerfil.

I vår dator används ett minne som i princip är ett RAM men som kallas REGISTER FILE (file är en engelsk term som bl a har betydelsen att lägga in dokument i en pärm eller i ett dokumentskåp). Kretsen innehåller 4 ord om vardera fyra bitar och heter 74170. Fig 3.35 visar en FILE, omfattande fyra kretsar, som lagrar A-, X-, P- och S-registren i ALPHA LSI.

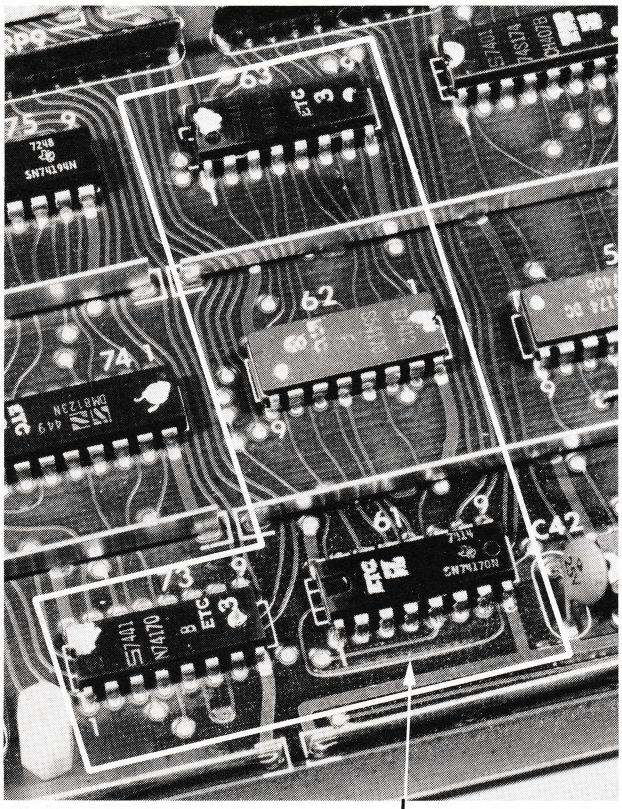


Fig 3.35 Registerfil (REGISTER FILE) för lagring av A-, X-, P- och S-registren i ALPHA LSI.

Innehållet i en kapsel av typ 74170 motsvarar ca 100 grindar. Det består av 16 vippor och har separata adressingångar för skrivning och läsning. Fig 3.36 visar innehållet i 74170.

I stora integrerade RAM av halvledare får man inte plats med vippor i krysspunkterna. Man använder då istället MOS-transistorer där informationen lagras i form av en liten laddning (några pC) på styrelektroderna. Sådana minnen kallas dynamiska eftersom laddning måste fyllas på (för att kompensera ofrånkomlig läckning). Även dynamiska minnen fungerar emellertid i princip på det sätt som beskrivits i fig 3.34.

RAM som innehåller vippor som minneselement kallas statiska halvledarminnen.

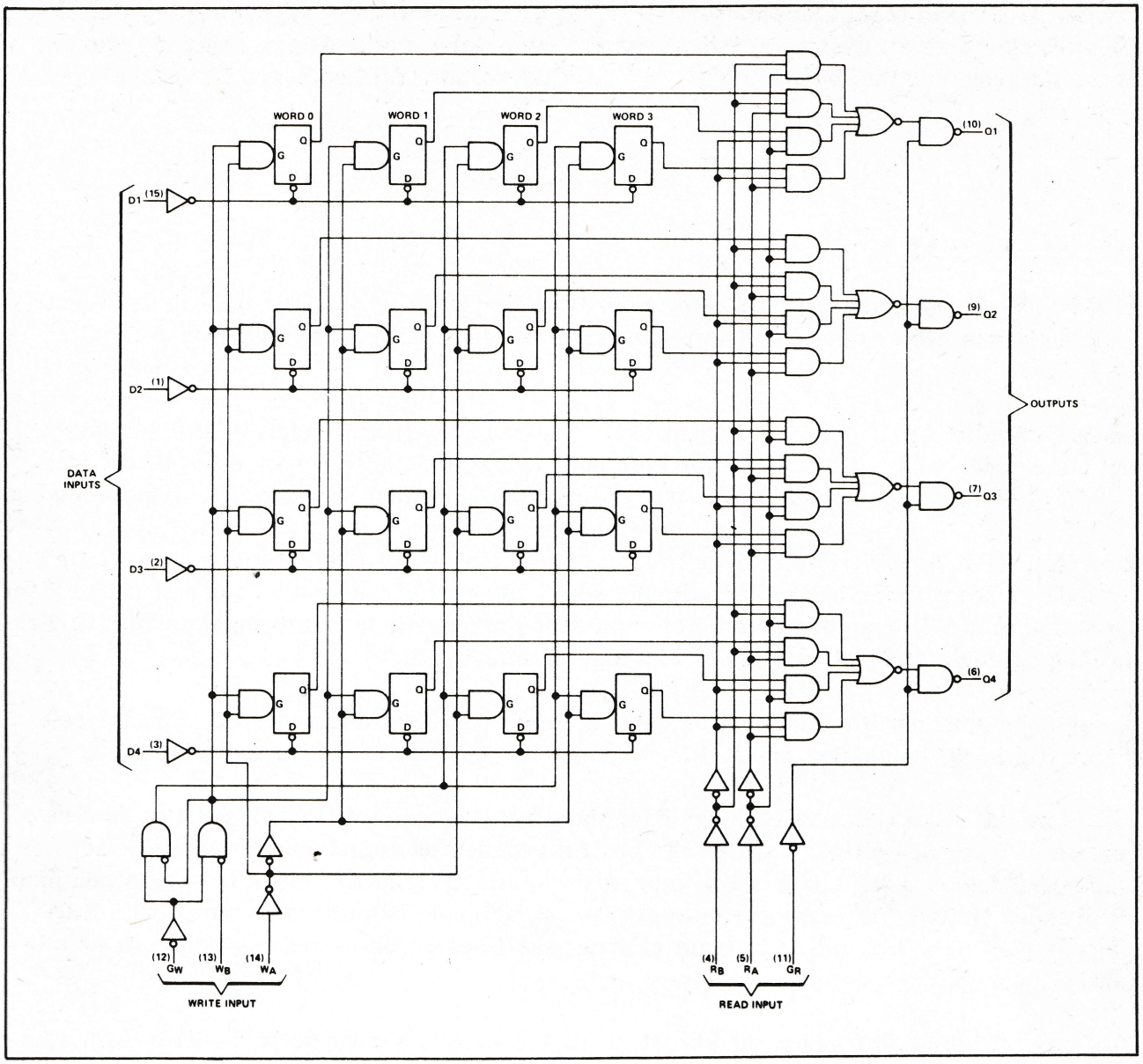


Fig 3.36 Innehållet i registerfilen 74170

3.2.3 ATT FÖRFLYTTA DATA

En räknedosa eller kalkylator arbetar som tidigare nämnts i serieform. Data kan därför "transporteras" längs en enda ledning i form av pulssekvenser.

En dator måste arbeta snabbt och då måste alla bitarna i ett ord förflyttas samtidigt - dvs i parallellform.

Vår dator arbetar ju med ord om 16 bitar. Det krävs alltså 16 parallella ledningar för att förflytta data från ett register till ett annat. Det blir därför gott om ledningar i en dator!

För att överhuvud taget klara av förflyttningen av data mellan alla olika ställen i en dator använder man ett system av "motorvägar" som delas mellan flera enheter i datorn. Dessa datavägar kallas bussar och vi ska nu studera hur trafiken styrs på bussystemet i vår dator.

3.2.3.1 En buss

ALU:n i vår dator placerar resultatet av sitt beräkningsarbete på 16 utgående ledningar. På kretskortet bildar dessa 16 ledningar resultatbussen eller R-bussen som den kallas, fig 3.37a.

ALU:ns resultat skall vid olika operationer tillföras M- eller W-registren, I-registret, den yttre data-bussen (D-bussen) för kommunikation med omgivningen eller latchen till A-, X-, P- och S-registren. Det måste tydligen finnas minst fem utgångar från R-bussen.

Ett sätt av vid önskad tidpunkt överföra data från R-bussen till ett register visas i fig 3.37b. 16 AND-grindar bildar här en signalgrind som kan styras från kontrollingången (2). I det ögonblick man vill överföra data höjer man spänningen på kontrollingången under tillräcklig lång tid för att registrets vippor ska hinna ställa in sig.

Normalt är grindnät för inläsning av data inbyggda i integrerade register. Vårt M-register (fig 3.29) innehåller en CLOCK-ingång som just är avsedd för detta ändamål.

Vill vi nu vid olika tidpunkter slussa över data från R-bussen i fig 3.37a till de fem registren behöver vi tydligen bara ge lämpliga styrorder på registrens kontroll- (eller klock-) ingångar. I fig 3.37a har vi markerat dataflödet från R-bussen till registren med pilförsedda linjer. Varje pil symboliserar en signalgrind och siffran anger kontrollingångens nummer. Det gäller ju (som vi strax skall se) att hålla ordning och reda på alla kontrollingångar i en dator.

ALU:ns utgångar är försedda med kraftiga utgångssteg och det är alltså ALU:n som ständigt bestämmer spänningarna på R-bussens 16 ledningar.

Man skulle kanske tro att datorkonstruktören skulle lägga ut R-bussen på kretskortet som en väg av parallella och raka ledningar. Fig 3.38 visar en del av vårt CPU-kort i genomlysning. Bland mängden av ledare som slingrar sig fram mellan ALU:n och de fem mottagarna (latchen, M- och W-registren, I-registret och drivsteget till yttre databussen) gömmer sig R-bussens 16 ledningar. Kretskortet har ledarmönster på båda sidor, på ena sidan går ledarna huvudsakligen horisontellt och på andra sidan vertikalt.

Frånsett svårigheten att få plats med alla ledare på kretskortet, är det tydligen ganska enkelt att sända ut data från en sändare, via en databuss och till ett flertal mottagare. Det är däremot inte fullt lika enkelt att köra in data från flera sändare till samma buss.

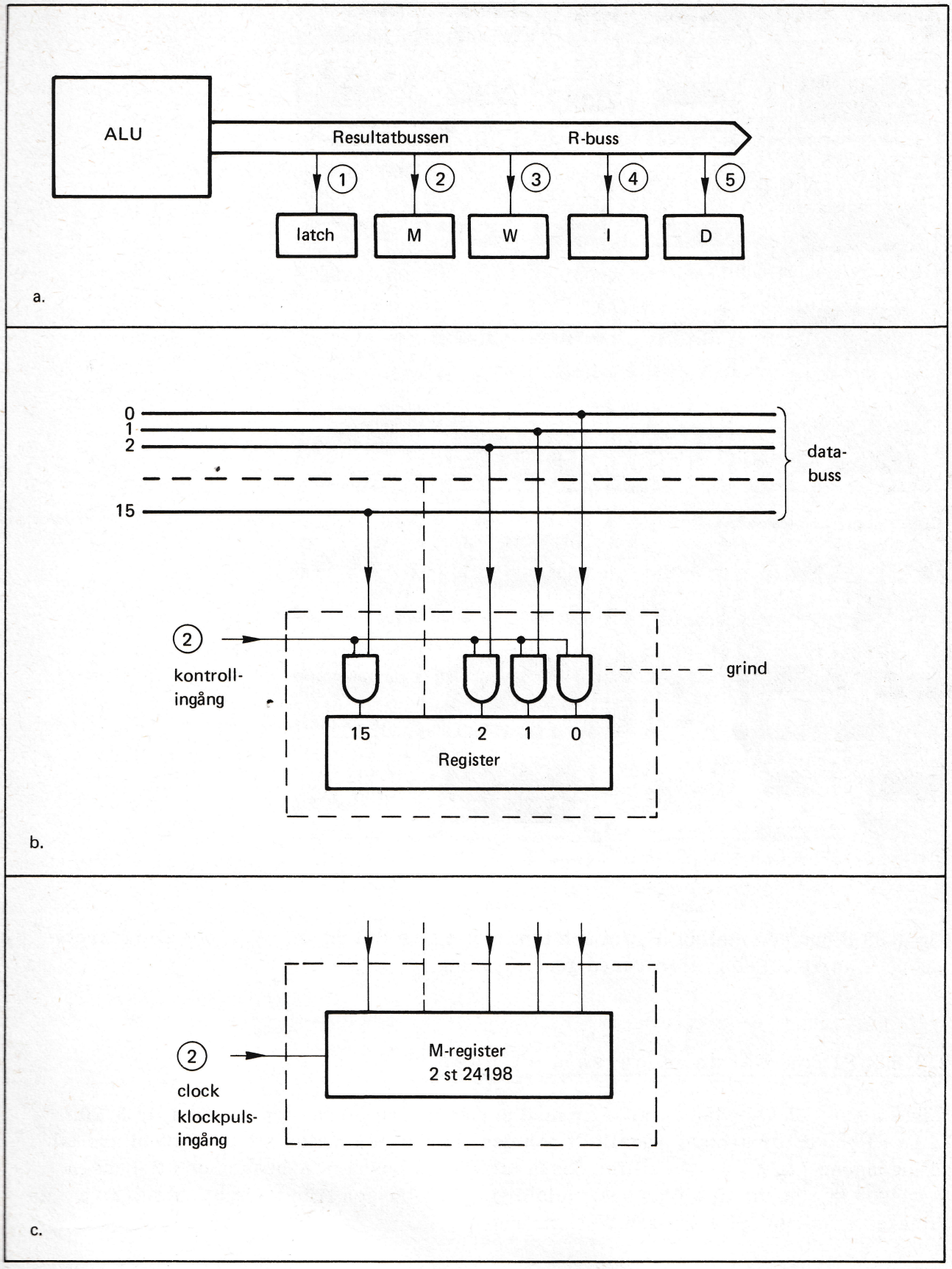


Fig 3.37 R-bussen från ALU:n.

Fig 3.38 R-bussen mellan ALU:n och fem mottagare (latch, M, W, I och D) på kretskortet. Fotograferat med genomlysning.

3.2.3.2 Flera sändare på samma buss

Räknedosan och kalkylatorn arbetar med serieaddition av den typ vi sett i fig 3.20. ALU:n i vår dator arbetar parallellt och den skall alltså matas med två 16-bitars tal på ingången. Fig 3.39a. Vi tillför dessa tal via två bussar, A-bussen och B-bussen. A-bussen kan matas bl a från ackumulatorn och B-bussen från flera av de tidigare omnämnda registren, bl a M- och W-registren.

Signalerna på ALU:ns resultatbuss bestäms enväldigt av ALU:n. Men signalerna på ingångsbussarna till ALU:n skall tydligen kunna styras från flera källor. Ett vanligt sätt att arrangera detta brukar kallas "matning med öppen kollektor" och vi ska se hur det fungerar.

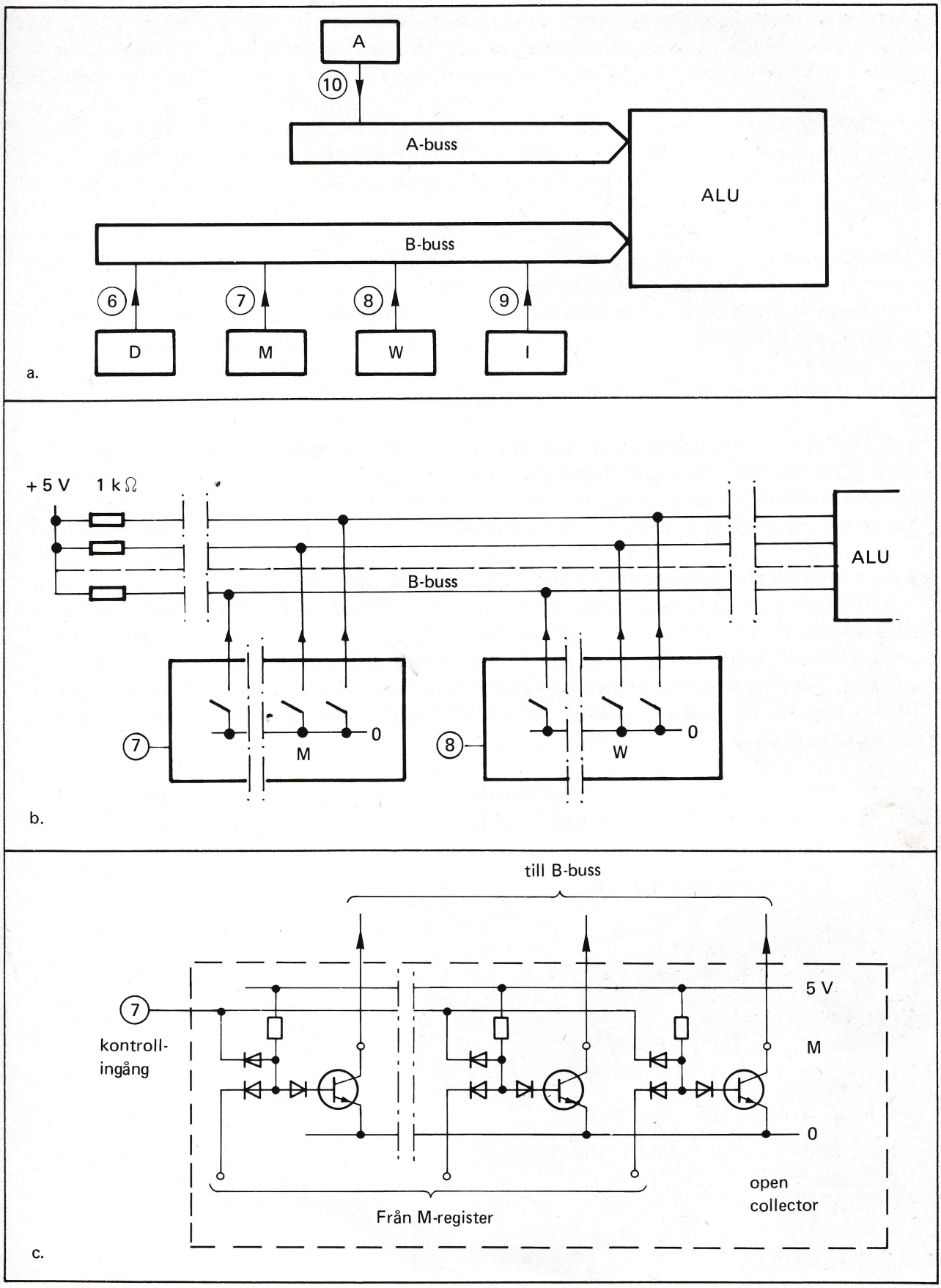


Fig 3.39 a. ALU:ns ingångsbussar
b. Principen för två sändare på samma buss
c. Detaljschema

Vi har egentligen redan bekantat oss med en buss som styrs av flera källor. I fig 3.33a utgör bitledningarna en buss som kan styras av var och en av de åtta ordledningarna - men enbart av en i taget. Funktionen innebär att önskade ledningar jordas via dioder.

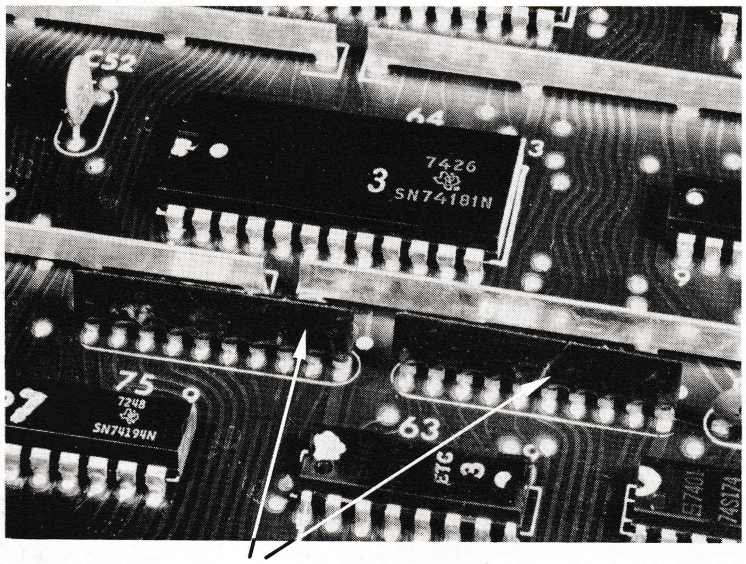
I fig 3.39b tillämpas samma princip. Samtliga ledningar i B-bussen ligger höga (+ 5 V) eftersom de matas via "pull up"-motstånd (på 1 kΩ) från matningsspänningen + 5 V. På bilden ser vi M- och W-registrens utgångar, och de har samma funktion i fig 3.39b som ordledningarna och dioderna har i fig 3.33.

Vi får alltså enbart låta ett register i taget styra B-bussen och detta kan ske via transistorer vars kollektor är direkt inkopplade till bussen, fig 3.39c. Normalt är dessa transistorer strypta och därför helt isolerade från bussen (Hade man haft kollektormotstånd inkopplade mellan transistorernas kollektorer och matningsspänningen, så hade dessa motstånd legat parallellt med "pull up" - motstånden och utgjort en belastning på bussen. Därför använder man kretsar av typ "öppen kollektor".)

I fig 3.39c ser vi hur utgångsdata från M-registret överförs till B-bussen då kontrollingången (7) läggs hög. Normalt ligger denna kontrollingång låg (0 V) och då är samtliga transistorer strypta (oavsett registrets inställning). Observera att det är samma enkla grind som i fig 1.4b vi använder - enda skillnaden är att den saknar kollektormotstånd.

För matning av bussar finns det speciella utgångskretsar som kallas tristate (tre tillstånd). I dessa kan utgången läggas hög, låg eller frikopplas helt. Tack vare att tristateutgången kan driva ström såväl i läge hög som låg blir den snabb. (De grindar av typ öppen kollektor vi visat i fig 3.39c kan endast driva ström vid låg utgång). Om flera tristate-grindar av misstag samtidigt får mata en buss så kan de bli förstörda (om en vill ligga hög och en annan samtidigt låg). Vid "öppen kollektor"-matning föreligger ingen sådan risk.

Fig 3.40 visar "pull-up"-motstånden för B-bussen till ALU:n. De har integrerats, 9 st i samma platta kapsel, och sitter anslutna direkt över B-bussen på kretskortet.



Pull up motstånd

Fig 3.40 Pull-up-motståndaren till B-bussen.

ALU:ns A-buss matas från fyra register (A, X, P och S), men eftersom dessa ligger i en FILE som adresseras (fig 3.36) föreligger här ingen risk för att två register samtidigt ska avge data till A-bussen.

Vi har nu sett hur data kan förmedlas via bussar och hur vi med kontrollingångar både kan tappa av data från bussarna eller tillföra data till bussarna. När det gäller att tillföra data till en buss, får man se upp så att enbart en sändare i taget får kommando över bussen.

I fig 3.5 finns en "styr- och räkneenhet". Vi har nu sett vad som döljer sig bakom begreppet räkneenhet - det är vår ALU med dess bussar. Styrenheten har ansvaret för "trafiken" på dessa bussar och det sker via kontrollingångar ("trafikljus") av den typ vi sett exempel på i fig 3.37 och 3.39.

3.2.3.3 I/O-bussen

Det stora CPU-kortet i vår dator kommunicerar med omgivningen via en buss som kallas MAXIBUS. (Olika datorfabrikanter ger sina bussar olika namn, i datorn PDP 11 kallas exempelvis motsvarande buss UNIBUS).

Maxibussen består av 58 ledningar som är uppdelade på tre grupper:

en adressbuss (A-buss) om 16 ledningar
en databuss (D-buss) om 16 ledningar
en kontrollbuss (C-buss) om 26 ledningar

Maxibussen framgår schematiskt av fig 3.41a.

A-bussen är en adressbuss med vars hjälp CPU:n lägger ut adressen till den enhet eller det minnesfack som CPU:n vill komma i kontakt med. A-buss är vedertagen benämning för adressbuss och tyvärr används samma namn även för den interna ackumulatorbussen i ALPHA LSI. Som användare kommer man ju inte i kontakt med inre A-bussen och därför är risken för förväxling inte så stor. Vi får hålla isär A-bussarna i fortsättningen och vid risk för förväxling tillägga "I/O" när det gäller den yttre bussen och "ackumulator" när det gäller den inre bussen.

A-bussen (I/O) har alltså 16 ledningar och via A-bussen kan därmed CPU:n adressera inte mindre än 2^{16} = 65536 olika enheter eller minnesfack.

D-bussen är en databuss som förmedlar 16-bitars ord från CPU:n till minne eller yttre enhet (eller i omvänd riktning). D-bussen är dubbelriktad och därmed är det väsentligt att dataflödet på D-bussen styrs från CPU:ns styrenhet. Man säger ibland att en sådan D-buss körs i "multiplex" med hänsyn till input och output.

C-bussen är en kontrollbuss om 26 ledningar. 18 av dessa förmedlar kommandon från CPU till yttre enheter och minnen. De övriga 8 utgör svarsledningar från yttre enheter och minnen till CPU:n.

CPU-kortet, ett antal minneskort och ett eller flera I/O-kort sitter alltså anslutna till

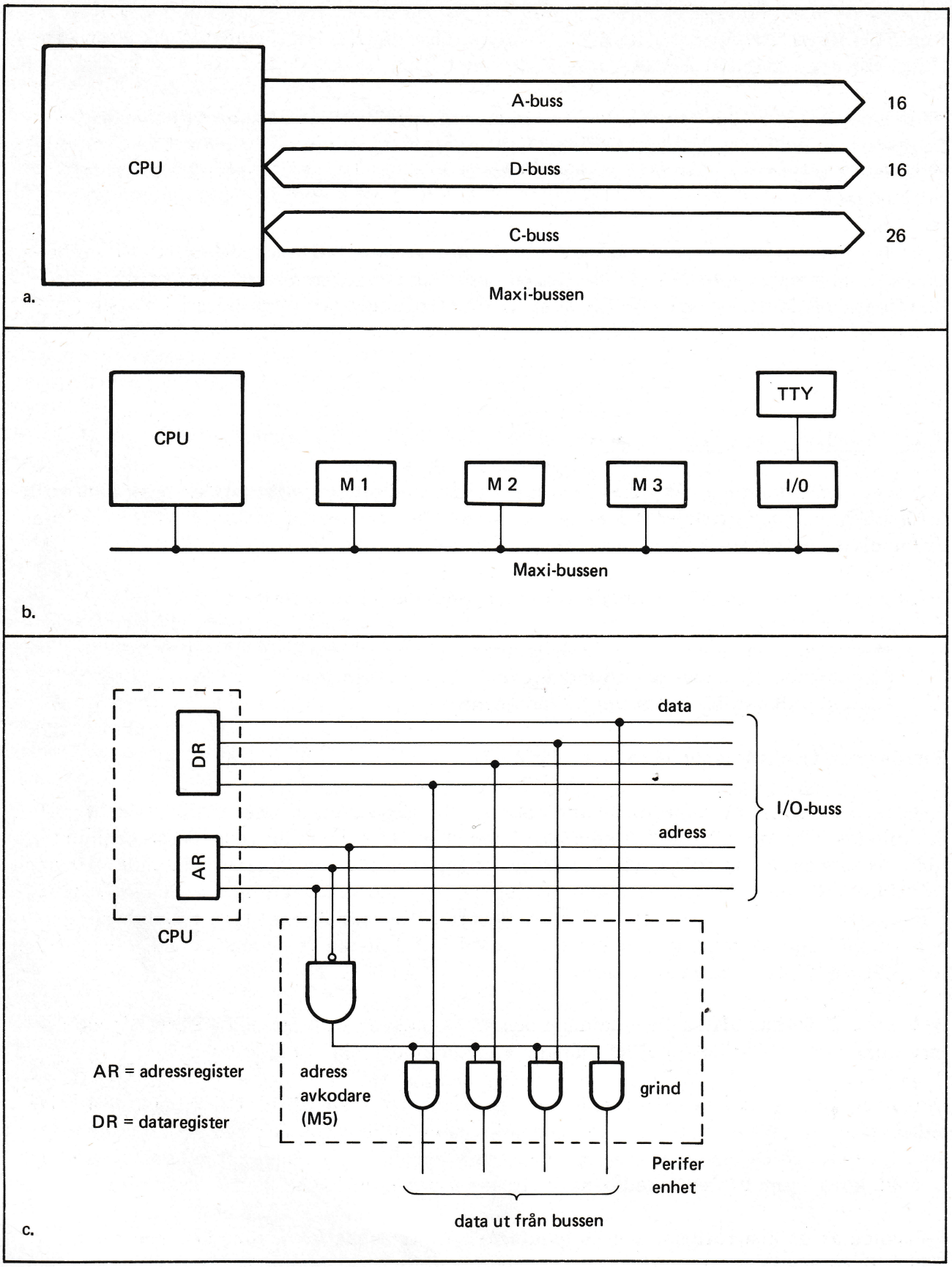


Fig 3.41 a. Maxibussen i ALPHA LSI
b. Arkitektur. M1 - M3 är minneskort. I/O är interface (exempelvis till TTY)
c. Principen för adressering och överföring av data till en perifer enhet.

en och samma buss, maxibussen eller I/O-bussen. Schematiskt tecknas detta så som visas på fig 3.41b.

CPU:n bestämmer vilken enhet som skall inkopplas till den gemensamma C- och D-bussen genom att lägga ut en adress på A-bussen. Principen framgår av fig 3.41c. Adressen på A-bussen avkodas av den adresserade enheten som därvid inkopplas via en grind till C- och D-bussen. Via C-bussen (som utelämnats i fig 3.41c) ger CPU:n order exempelvis om läsning eller skrivning och fig 3.41c visar det fall att data överförs från CPU:n och läses av den adresserade enheten.

Olika datorer har olika "arkitektur". Vissa datorer har en minnesbuss och en I/O-buss. Andra har en gemensam buss för både minne och yttre enheter. Vår ALPHA LSI har en gemensam "maxibuss" för både minne och I/O-enheter. Adresseringen av en minnescell eller en I/O-enhet sker därför i princip på samma sätt.

Alla kretskort i vår dator (CPU, minneskort, I/O-kort och kraftförsörjning) är anslutna till ett gemensamt "moderkort" som innehåller maxibussen. Fig 3.42a.

Maxibussen är tillgänglig i alla uttag på moderkortet. Det spelar därför i princip ingen roll var CPU:n, minneskort och I/O-kort placeras. Det ingår emellertid en del extra ledningar till CPU:n och den skall därför alltid placeras överst i ALPHA LSI. CPU-kortet är ju dessutom tjockare än minneskorten och måste därför ha större utrymme. Fig 3.42b.

De kretsar som på CPU:n matar den yttre A-bussen (I/O) och de utgående linjerna på C-bussen kallas DRIVERS (drivsteg) och ligger omedelbart intill kretskortets anslutningskontakter. Fig 3.43.

D-bussen arbetar i båda riktningar och de kretsar på CPU:n som är anslutna till D-bussen är av typ DRIVER/RECEIVER, dvs både sändare och mottagare (beroende på kontrollingångens ställning).

Både A-bussens drivsteg och D-bussens sändare/mottagare har tristate-utgångar.

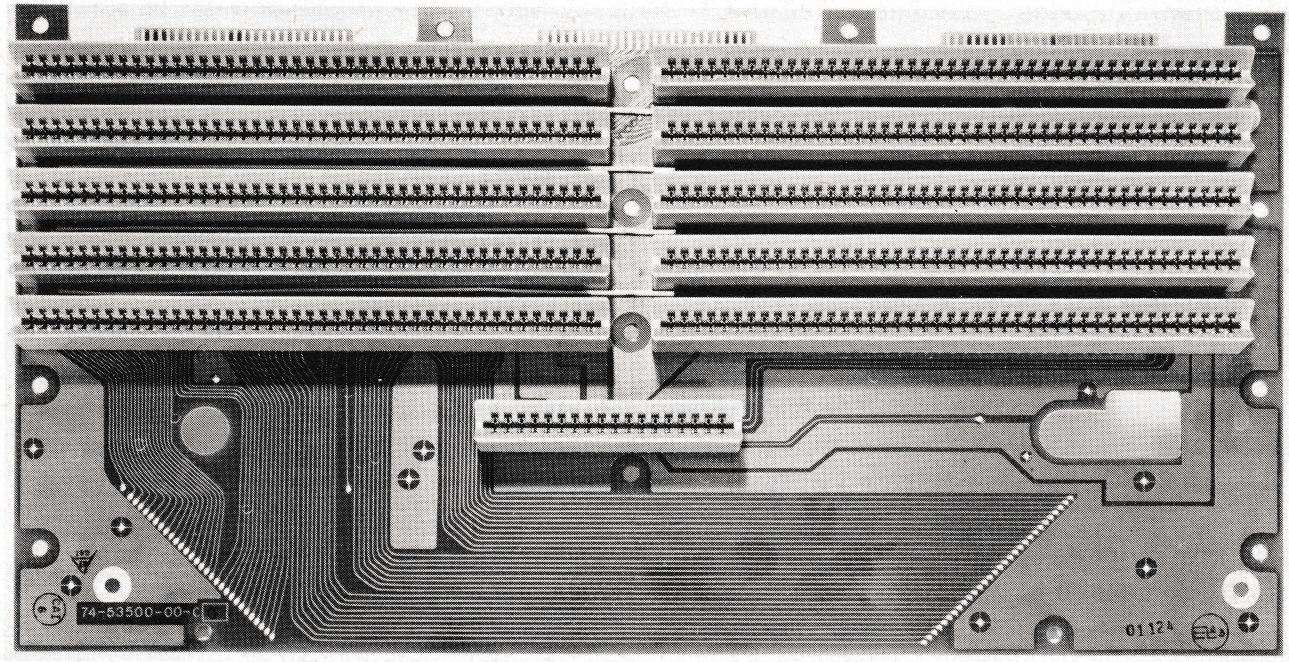


Fig 3.42 a. Moderkortet

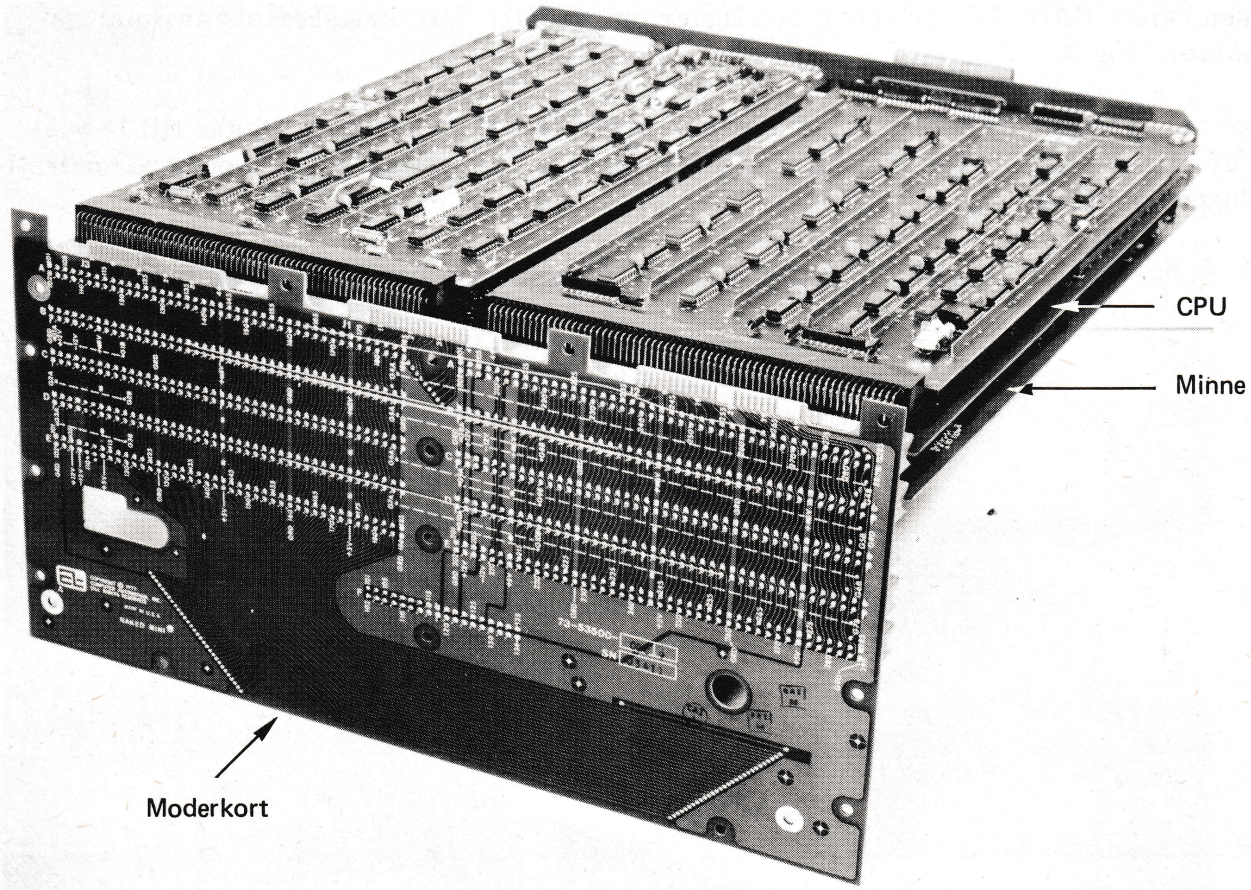


Fig 3.42 b. Moderkort med CPU och minneskort.

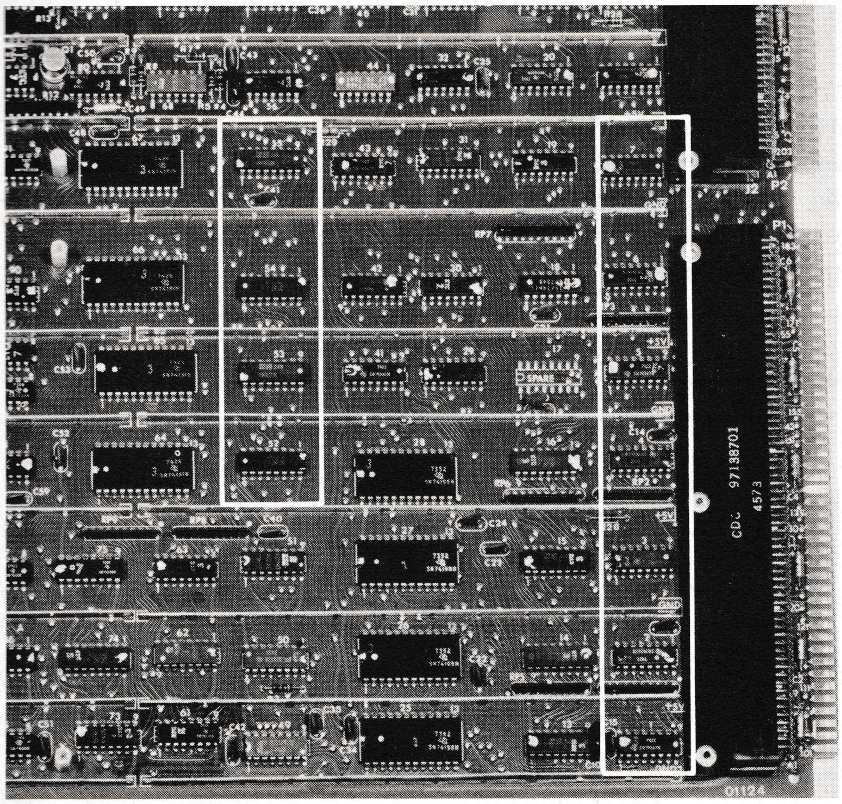


Fig 3.43 CPU:ns kretsar för sändning och mottagning på maxibussen.

3.2.4 STYRENHETEN

Nu har vi kommit fram till den intressantaste delen i vår dator, styrenheten. Den fungerar som en trafikcentral och håller i alla trådarna i det omfattande trafiknät, som en dator utgör.

Från styrenheten läggs en "grön våg" av trafikljus mellan en minnescell och instruktionsregistret när datorn hämtar sin nästa instruktion.

På order av styrenheten tolkas instruktionen och startas önskade subrutiner i mikroprogrammet. Om det är en LDA-instruktion läggs en ny "grön våg" från önskad minnescell till ackumulatorn. Om det är en STA-instruktion läggs en motsvarande våg men i omvänd riktning.

För att kunna följa trafiken måste vi nu ha en karta över vägnätet. I fig 3.44 ser vi en sådan karta. Den ger en förenklad bild av vår ALPHA LSI 2 men den är fullt tillräcklig för att ge oss en rättvisande beskrivning av funktionen.

Vi ska nedan först beskriva blockschemat i fig 3.44 och sedan följa förloppet i schemat när vi exekverar vårt tidigare "demonstrationsprogram".

3.2.4.1 Blockschemat

De flesta datorfabrikanter hävdar att deras maskiner har en optimal och överlägsen arkitektur. Det finns många sätt att bygga en minidator och vid blockschemats sammansättning får man göra en rad kompromisser. Det är inte vår uppgift att här väga för och nackdelar mot varandra.[x)] Fig 3.44 visar blockschemat för ALPHA LSI och med hänsyn till den stora mängd tillämpningar i vilka man använt ALPHA LSI, så måste detta blockschema inrymma förnuftiga kompromisser i olika avseenden.

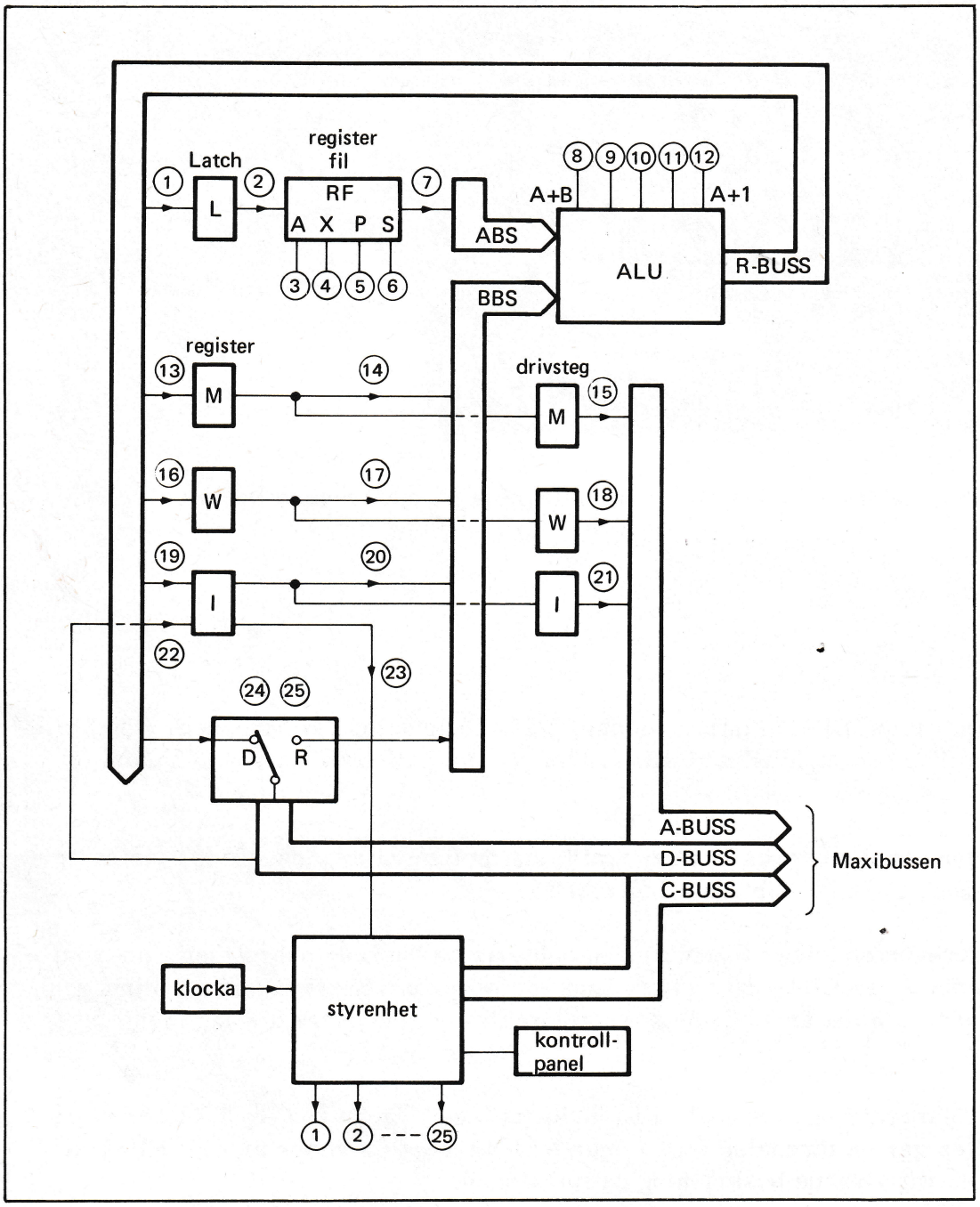


Fig 3.44 Blockschemat för ALPHA LSI 2.

---

x) Läs gärna boken "The value of power" från General Automation som ett exempel på hur man kan argumentera för en viss arkitektur.

Längst upp till höger ser vi ALU:n som matas från ackumulator- och B-bussarna (som vi här betecknar ABS och BBS för att undvika förväxling med maxibussens A-buss).

På kontrollingångarna (märkta 8 9 10 11 och 12) får ALU:n order om vilken operation den skall utföra och den placerar resultatet därav på resultatbussen (R-bussen).

Resultatet på R-bussen kan nu överföras till följande fem enheter: (som vi redan tidigare lärt känna i avsnitt 3.2.3.1)

en latch, genom kontrollingång 1

M-registret, genom 13

W-registret, genom 16

I-registret, genom 19

D-bussen (del av maxibussen), genom 24

Ackumulatorbussen ABS matas från registerfilen AXPS genom 7. Adressen till önskat register läggs på adressingångarna 3, 4, 5 eller 6.

Registerfilen matas från latchen via 2 och latchen får, som vi sagt tidigare, sin information från R-bussen via 1.

B-bussen (BBS) matas från följande källor:

M-registret, via 14

W-registret, via 17

I-registret (pos 0 - 7), via 20

D-bussen (receive), via 25

Maxibussen går ut till höger i fig 3.44. Den består av tre delar:

adressbussen (A-bussen)

databussen (D-bussen)

kontrollbussen (C-bussen)

Adressbussen kan matas från tre källor:

M-registret via ett drivsteg (M-driver) och 15

W-registret via ett drivsteg och 18

I-registret (pos 0 - 7) via ett drivsteg och 21

Adressbussens ställning bestäms alltid av CPU:n och den är därför enkelriktad (ut från CPU:n).

Databussen har dubbelriktad trafik, men givetvis enbart i en riktning i taget (multiplex). Databussen riktas utåt (från CPU:n) som sändare (driver) på order från kontrollingång (24). Databussen riktas inåt (mot CPU:n) som mottagare på order från kontrollingång (25).

I-registret kan via en direktingång, (22), hämta data direkt från D-bussen. De extra in- och utgångarna till I-registret är utförda med hjälp av grindar som kallats I-select i fig 3.38.

ALU:n kan tillsammans med registren utföra en rad logiska och aritmetiska operationer. Vi kommer emellertid här enbart att utnyttja två av dessa nämligen addition (R = A + B) och inkrementering (dvs öka med 1, R = A + 1). Kontrollingångarna för dessa funktioner kallas (8) och (12) i fig 3.44.

Nu har vi alltså lärt känna trafiknätet i ALPHA LSI. Det är styrenhetens uppgift att dirigera trafiken - dvs de 25 signalljusen - så att vi får önskade operationer utförda. Vi ser styrenheten längst ner i fig 3.44. Styrenheten i fig 3.44 har 25 utgående kontrollledningar (I verkligheten är det 45 st men vi har förenklat schemat för att lättare kunna beskriva förloppen.)

3.2.4.2 Styrenhetens uppbyggnad

När vi skrivit ett program, placerat det i minnet, ställt in programräknaren på startadressen och tryckt på startknappen - ja då tar styrenheten över ansvaret för det fortsatta arbetet. Styrenheten ser till att den första instruktionen hämtas på den utpekade platsen i minnet och placeras i instruktionsregistret.

Fig 3.45 visar situationen just när instruktionen hamnat i instruktionsregistret. Styrenheten öppnar grind (23) och släpper in operationskoden. Denna tolkas och startar en serie mikroinstruktioner som tillsammans utför den önskade instruktionen.

Hur kan man nu bygga upp en styrenhet med så speciella egenskaper? Vi ska nedan bekanta oss med tre olika lösningar.

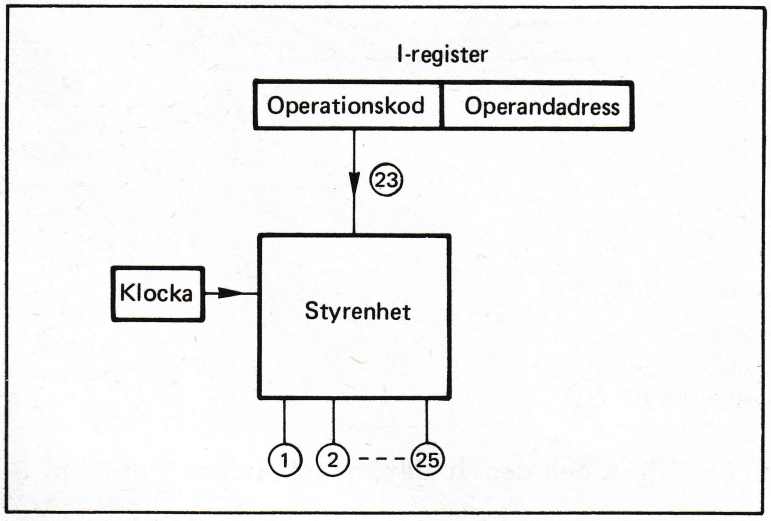


Fig 3.45 Styrenhetens uppgift är att hämta och tolka instruktionen och därefter att utföra den.

3.2.4.2.1 Styrenhet med grindnät

Ett grindnät (eller kombinationskrets) kan byggas upp av grindar av de enkla typer vi sett i fig 1.4b och 1.8. Halv- och heladderarna i fig 3.18 och 3.19 är exempel på sådana grindnät. Det karakteristiska för alla grindnät är att en given insignalkombination alltid ger en entydigt bestämd kombination av utsignaler.

Att hämta och exekvera en instruktion är som regel ett omfattande arbete som måste ske i flera etapper. Det krävs alltså flera etapper av spänningskombinationer på de från styrenheten utgående kontrolledningarna.

Ett sätt att bygga upp styrenheten är därför att konstruera ett grindnät som dels tar emot i tiden uppdelade "etappsignaler" och dels tar emot operationskoden från instruktionsregistret. Etappsignalerna genereras av en "timing unit" exempelvis i form av åtta ledningar som i tur och ordning läggs höga i takt med inkommande klockpuls. Vi får på detta sätt åtta i tiden uppdelade etapper, alla med entydiga spänningskombinationer på kontrollutgångarna (1) till (25). Fig 3.46.

De första fyra etapper kan exempelvis lägga de signalvägar som krävs för hämtning av instruktion (på den plats i minnet som utpekas av P-registret eller programräknaren).

Under de fyra efterföljande etapperna är operationskoden från instruktionsregistret inkopplad till grindnätet och bestämmer därigenom tillsammans med etappsignalerna exekveringen av instruktionen.

Att bygga upp grindnät med så komplex funktion som krävs i en styrenhet är ingen enkel uppgift. När konstruktionen väl har gjorts och grindarna lötts upp på ett kretskort, kan man inte längre ändra funktionen. Funktionen ligger ju i grindnätets sammansättning.

Man kallar denna typ av uppbyggnad för fast logik, "wired logic" eller "random logic" eftersom funktionen bestäms genom hopkoppling av logikgrindar.

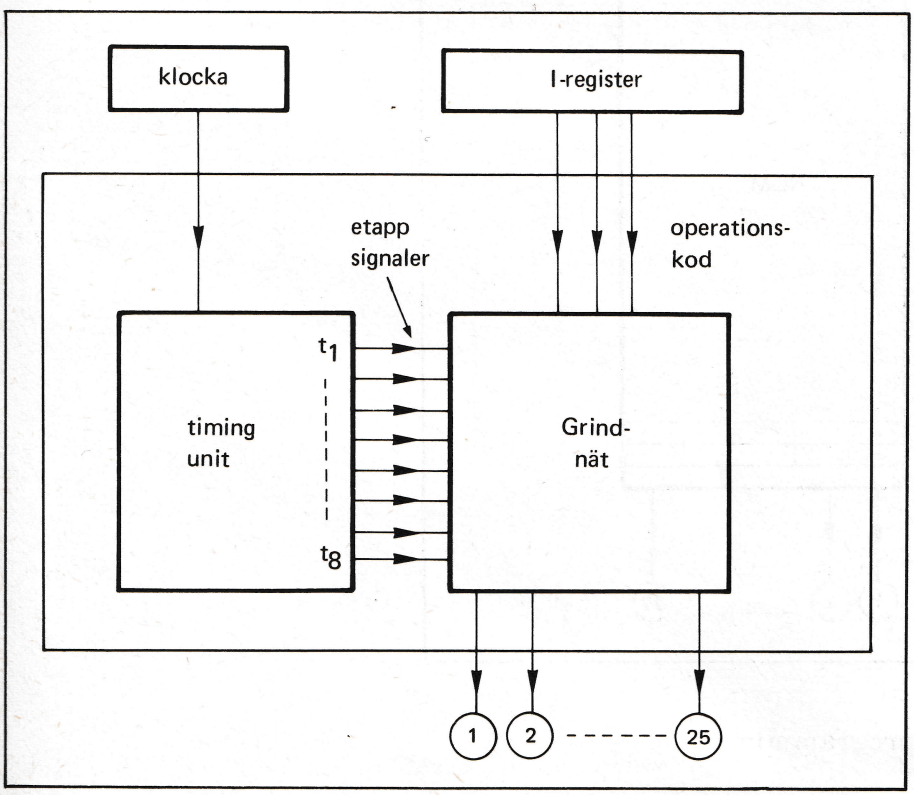


Fig 3.46 Styrenhet uppbyggd av grindnät.

3.2.4.2.2 Mikroprogrammerad styrenhet

Vi har använt termen "programmera" för inskrivning av data i ett läsminne eller ROM. Ett sätt att bygga en styrenhet är just att programmera ett läsminne med de etapper av kontrollsignaler vi ovan beskrivit. Ett ROM organiserat i ett antal ord om vardera 25 bitar skulle kunna användas för styrning av de 25 utgående kontrolledningarna i vår styrenhet i fig 3.44.

Varje instruktion (som i detta sammanhang ofta kallas maskininstruktion) måste i detta fall ha sina arbetsetapper lagrade i form av ett eget mikroprogram. Instruktionskodens uppgift blir här att tala om var i ROM:et, som den aktuella maskininstruktionen har sitt mikroprogram lagrat.

Fig 3.47 visar principen. Vi tänker oss här ett minne som omfattar 256 ord om vardera 27 bitar. 256 ord kräver (enligt fig 3.30) 8 adressledningar och dessa kommer i fig 3.47 från en räknare som räknar pulserna från klockan.

Kretsen i fig 3.47 fungerar nu precis som det programminne vi tidigare beskrivit i samband med kalkylatorn (fig 2.3 och 2.10). Om räknaren är nollställd får vi successivt ut

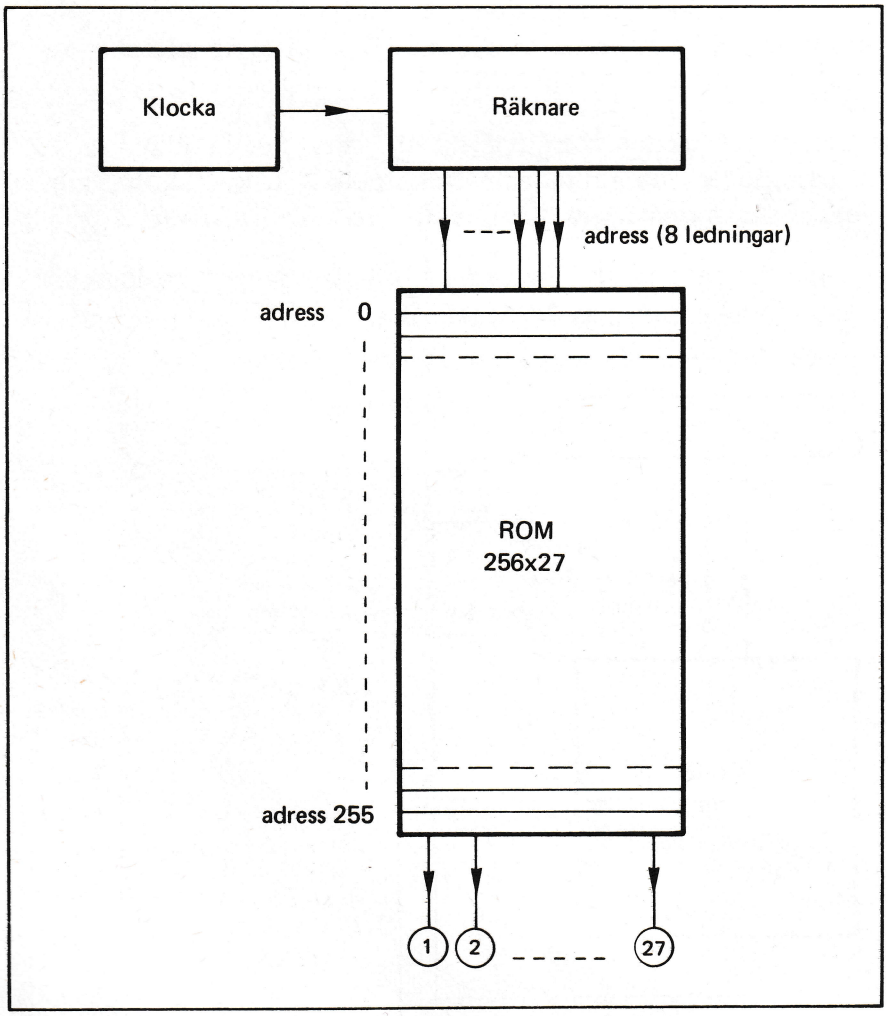


Fig 3.47 Principen för ett programminne.

samtliga ord (eller etappkombinationer) i ordningsföljd på utgångarna. Vi har emellertid lagt in två utgångar mer än vad vi behöver för styrningen av signalvägarna i fig 3.44. Och nu kommer knepet! Fig 3.48.

Vi har valt en räknare med två speciella egenskaper. Den kan nollställas med en CLEAR-ingång (precis som registret i fig 3.29). Den kan laddas parallellt (med en kontrollsignal kallad LOAD). Vi styr nu CLEAR-ingången med utgång (26) från minnet och LOAD-ingången med utgång (27). Vi kan på detta sätt styra räknaren under programkontroll från ROM:et.

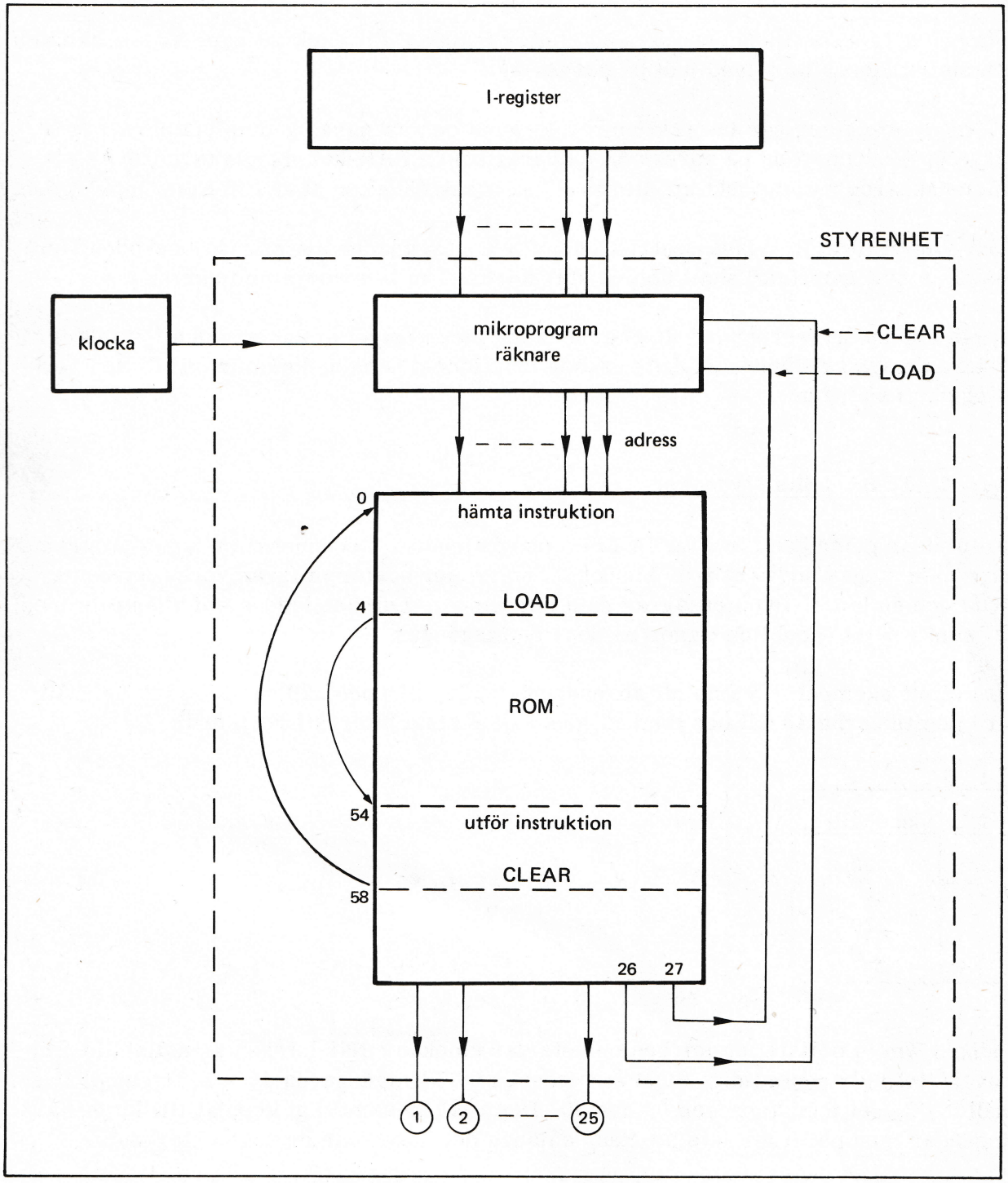


Fig 3.48 Styrenhet med ROM och mikroprogramräknare.

Fig 3.48 visar hur man kan lagra mikroprogram i ett ROM. Antag att det som tidigare åtgår fyra etapper för hämtning av programmet och fyra för exekveringen. Vi låter då hämtningsrutinen vara placerad i adresserna 0 till 3 och låter ett load-kommando (dvs 27 vara lagrat i adress 4. När hämtningen av instruktionen är fullbordad kommer därför räknaren att laddas med instruktionsregistrets operationskod. Operationskoden utgör ju ett binärt tal och när räknaren blir laddad med detta tal avger den detta tal som en adress till ROM:et. Är exempelvis instruktionskoden 110110 (dvs 54) så hoppar räknaren från 4 till 54. Ut från ROM:et kommer därvid det ord som ligger permanent lagrat på adress 54.

Vi antar nu att första mikroinstruktionen (eller etappen) för exekveringen av den aktuella maskininstruktionen är inlagd just på adress 54.

Efter fyra klockpulser har instruktionen fullgjorts och räknaren genomlöpt adresserna 54, 55, 56, 57. Om vi nu på adress 58 placerar ett CLEAR-kommando (dvs 26) så kommer räknaren automatiskt att återgå till noll och därefter hämta in nästa instruktion.

Det är, som vi förstår, vissa restriktioner för uppbyggnaden av instruktionskoden, om den på detta sätt samtidigt skall utgöra startadress i mikroprogramminnet.

Att bygga upp en styrenhet med ROM är å andra sidan betydligt enklare och flexiblare än att konstruera grindnät med den önskade funktionen. Men nu kommer vi till den verkligt eleganta lösningen.

3.2.4.2.3 ALPHA LSI:s styrenhet

Fig 3.49 visar principen för ALPHA LSI:s programverk. Det innehåller ingen programräknare men stegas ändå fram av klockan. Som vi ser består programverket enbart av ett ROM och en latch. ROM:et lagrar data permanent. Latchen laddas vid klockpuls med det tal som i detta ögonblick ligger på dess dataingångar.

Låt oss ta ett exempel och anta att adresserna 0, 30, 210 och 220 har följande innehåll lagrat i positionerna 49 till och med 56 (dvs i de 8 sista bitarna i resp ord):

adress	Innehåll
0	30
30	210
210	220
220	0

Vad händer om vi nollställer latchen och startar klockan? När latchen är nollställd (och har innehållet 0) har samtidigt ROM:et adressen 0. På utgångarna 49 - 56 ligger därvid innehållet i facket med adressen 0, dvs 30. Detta tal är samtidigt kopplat till latchens dataingångar men påverkar inte latchen, sålänge det inte kommer någon klockpuls.

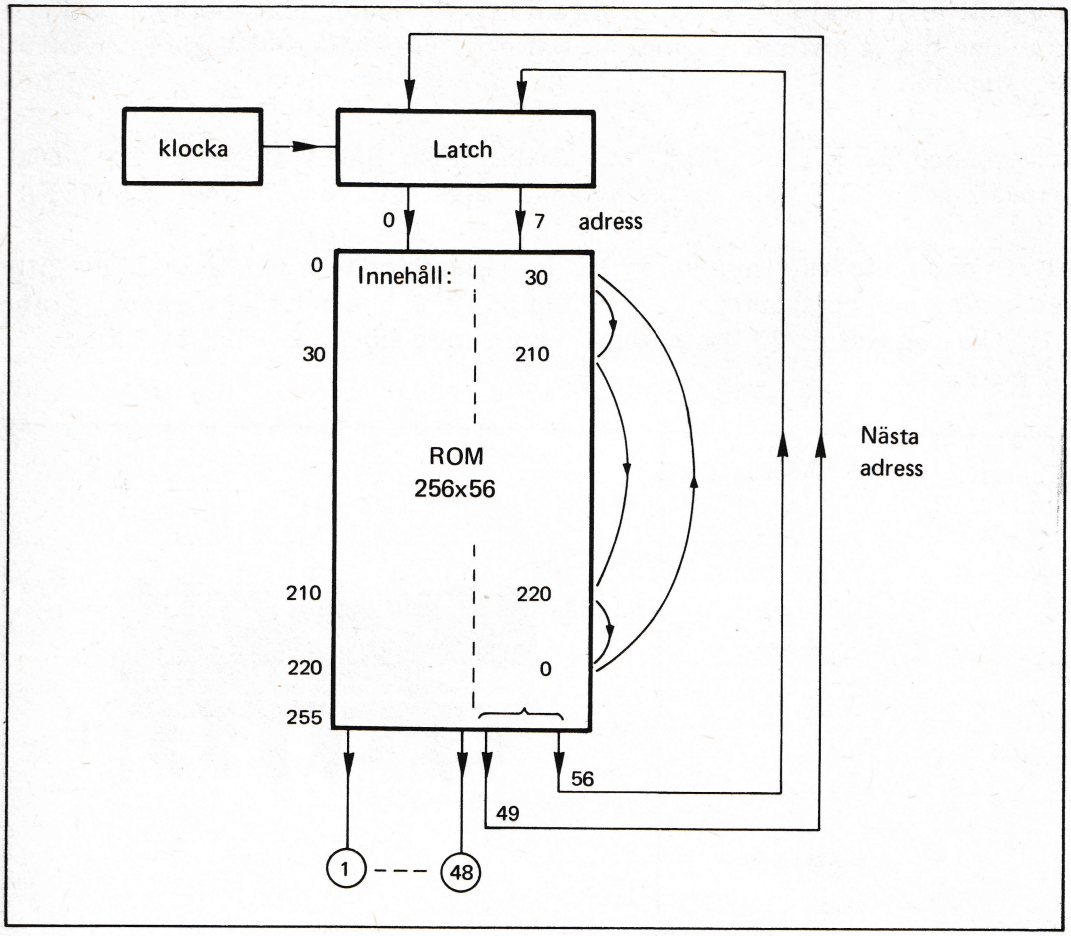


Fig 3.49 Självadresserande programverk.

När första klockpulsen anländer ställs alltså latchen i läge 30 (Man säger ibland att latchen laddas med talet 30). Latchen är så konstruerad att ingången omedelbart frånkopplas vid klockning. Nu får alltså minnet adressen 30 och utgångarna 49 - 56 presenterar talet 210 för latchens dataingångar (som ligger spärrade).

Vid nästa klockpuls hoppar vi alltså till adressen 210 och finner innehållet 220. Nästa hopp sker till 220 med innehållet 0.

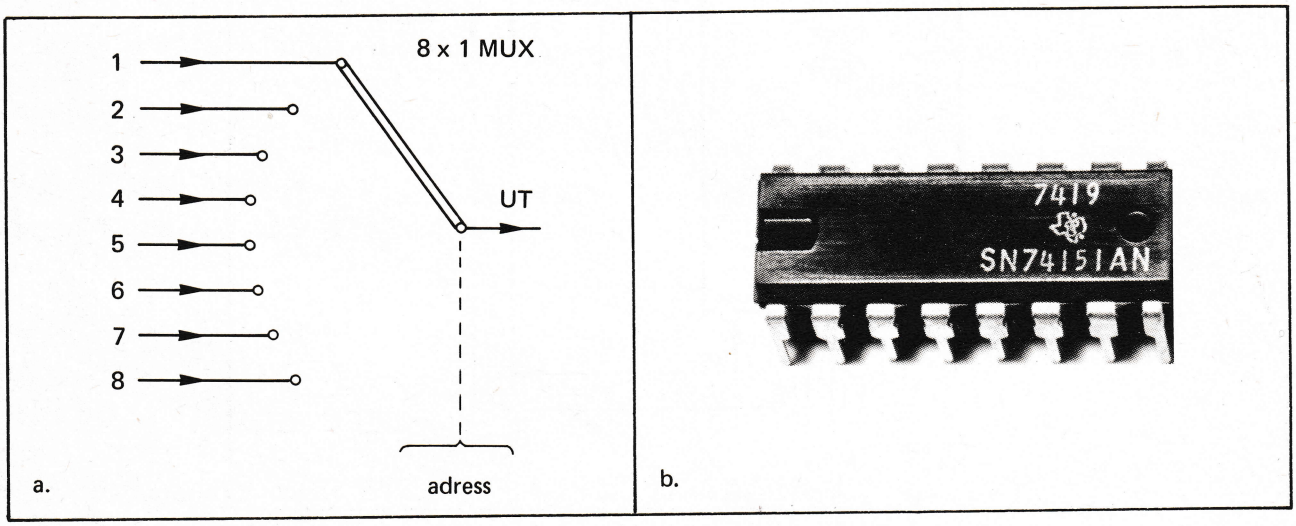
Efter fyra klockpulser är vi tillbaks till utgångsläget. Vi har hoppat igenom programminnet som en loppa. Vi kan programmera hoppen godtyckligt och på det viset kan vi utnyttja varje udda fack, som av någon anledning icke tidigare utnyttjats.

Programräknaren i fig 3.48 kräver två kapslar (4 bitars binärräknare i varje kapsel). Ett ROM på 256 ord om 8 bitar kräver också två kapslar (1024 bitar per kapsel). Med hänsyn till antalet kapslar går det alltså "på ett ut" om man använder mikroprogramräknare eller självadressering med "nästa adress" i minnet. Men med "nästa adress" lagrad i varje minnesfack får vi en flexibilitet som ger oss utomordentliga möjligheter både att använda lämpliga startadresser och att utnyttja samtliga minnesfack.

Nu ska vi se hur enkelt man kan lagra mikroprogrammen för maskininstruktionerna i ett programverk av den typ vi just har beskrivit. Det enda vi behöver tillägga är en MUX eller multiplexer. Fig 3. 50.

En MUX är en omkopplare och kallas ibland dataväljare. Principen framgår av fig 3. 50a. På lilla CPU-kortet finns en MUX som består av sex kretsar av typen 74151 (8 x 1 MUX).

Trafikcentralen i vår dator utgörs alltså av en MUX, en latch och ett ROM. De utgör tillsammans 23 plastkapslar på kretskortet. Vi ser dem på fig 3. 50c och blockschemat framgår av fig 3. 51. Här har endast de 25 kontrollutgångar som behövs för vårt förenklade schema i fig 3. 44 medtagits.



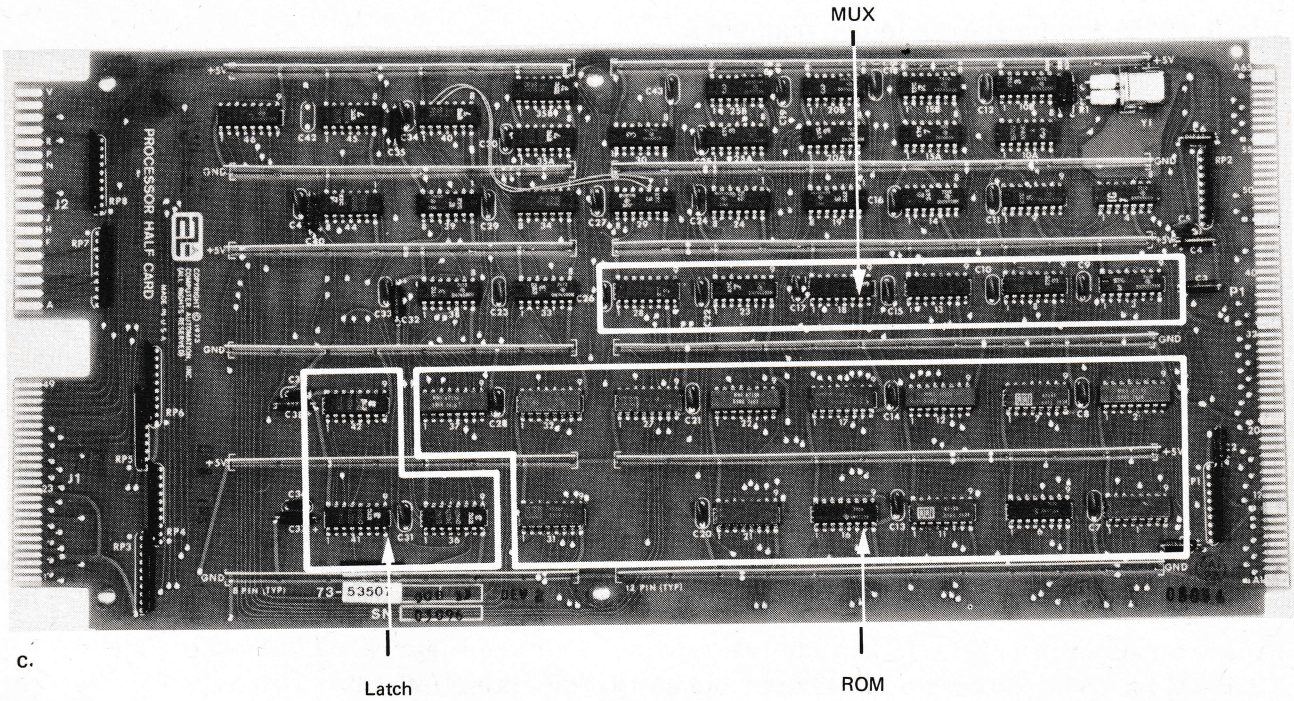


Fig 3. 50 a. Principen för en MUX
b. Kretsen 74151 (8 x 1 Mux)
c. MUX, latch och ROM - trafikcentralen i ALPHA LSI.

Den bild vi här tecknat av styrenheten är ytterligt förenklad. Vi bör kanske ta med ytterligare två block i vårt schema, kontrollbussens inkoppling och rutinen för avbrott. Fig 3.52.

En maskininstruktion skall först hämtas (fetch) och sedan exekveras (execute). Låt oss nu följa styrenhetens hantering av maskininstruktionen LDA.

Mikroinstruktionerna för hämtning är placerade med början från adress 0 i ROM:et. Sista mikroinstruktionen skall innehålla en order till MUX:en att slå om till läge I. Vid efterföljande klockpuls kommer därvid latchen att hämta nästa adress från instruktionsregistret. Instruktionskoden (LDA) tolkas nu som en adress och föranleder ett hopp i ROM:et. Den första order som är lagrad på denna adress måste nu vara omställning av MUX:en till läge N, dvs så att nästa adress hämtas från minnet på det sätt vi tidigare beskrivit i fig 3.49.

De efterföljande mikroinstruktionerna för exekveringen av LDA-instruktionen kan vara lagrade på godtyckliga ställen i vårt ROM. I fig 3.51 har vi lagt dem i en följd som slutar med angivande av nästa adress = 0. Efter avslutat arbete med en maskininstruktion hoppar alltså programverket automatiskt tillbaka till början och hämtar en ny maskininstruktion.

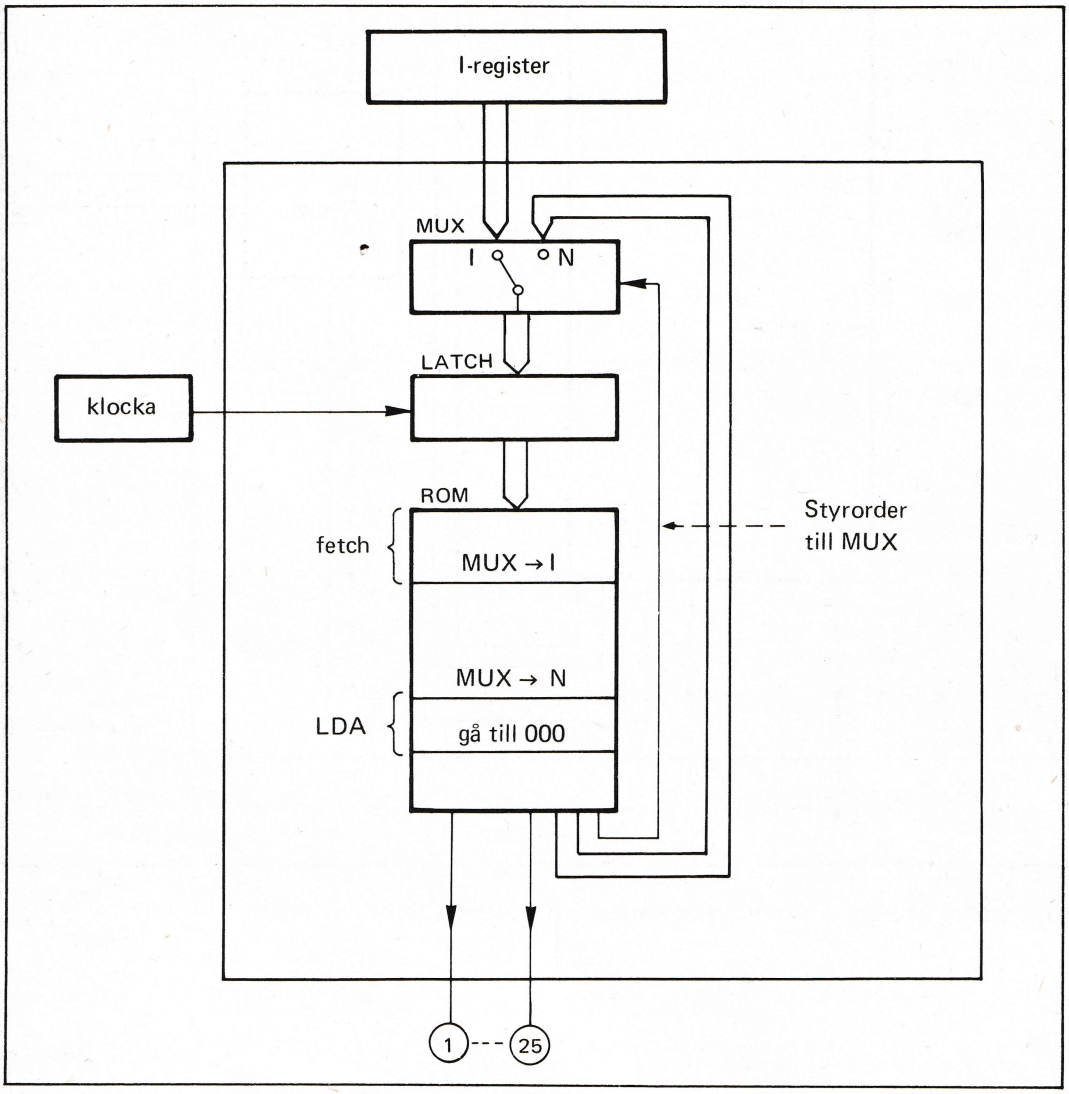


Fig 3.51 Styrenhet med ROM, MUX och latch. (MUX:en har två lägen, I = adress från I-register och N = nästa adress från ROM).

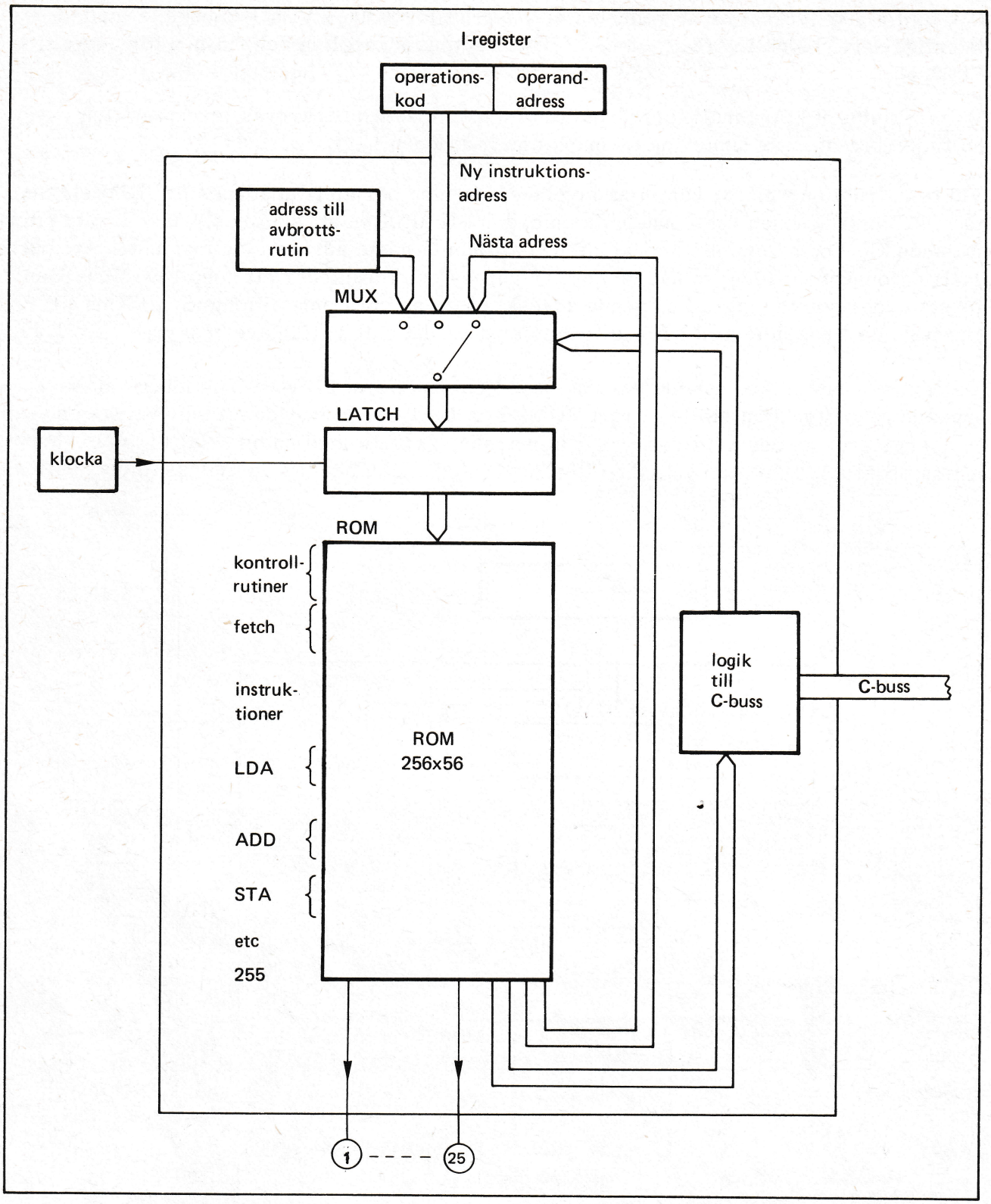


Fig 3. 52 Förenklat blockschema för styrenheten i ALPHA LSI.

Kontrollbussen (C-bussen) ska ju förmedla styrorder från CPU:n till minnen och I/O-enheter. Den ska även förmedla svar eller kvittens (acknowledge) från I/O-enheten tillbaks till CPU:n (exempelvis när en I/O-enhet har lagt ut önskade data på D-bussen). Kontrollbussen är därför i princip inkopplad på det sätt som visas i fig 3.52.

När en I/O-enhet vill kalla på CPU:ns tjänster, så lägger den ut en begäran om avbrott (IUR = interrupt request). Den begär med andra ord att CPU:n skall avbryta pågående arbete och istället ägna sig åt den anropande I/O-enheten.

I fig 3.52 har vi lagt in några kontrollrutiner vilka alltid avsöks innan ny instruktion hämtas. Med dessa mikroinstruktioner avkänner CPU:n om någon I/O-enhet anropar (med IUR). Om detta är fallet kopplas Mux:en över till ett tredje läge så att en adress hämtas från D-bussen. Denna adress är utlagd av den anropande I/O-enheten och anger var programmet för avbrottsrutinen finns lagrad i minnet.

Istället för att hämta nästa maskininstruktion för exekvering av det pågående programmet, så hämtar CPU:n nu första maskininstruktionen för I/O-enhetens avbrottsrutin. Den börjar vanligen med lagring av samtliga register i minnet och utför därefter den önskade I/O-rutinen.

När I/O-enhetens önskemål har uppfyllts, så laddas registren åter med det innehåll som de hade omedelbart före avbrottet. CPU:n kan därmed återgå till sitt tidigare arbete.

Vid alla minnesoperationer måste CPU:n vänta medan önskat ord plockas fram ur minnet. När vi hämtar ett ord från minnet, så läggs först ordets adress ut på A-bussen (I/O) och samtidigt signalen MST (dvs memory start, starta minnet) på C-bussen.

Kontroll-logiken stoppar nu klockan tills svaret MACK (dvs memory acknowledge, minnet klart) har anlänt från minnet till CPU:n via C-bussen. Det tar ca 1 μs för ALPHA LSI att göra en minnescykel (dvs tiden mellan MST och MACK). Som jämförelse utförs en mikroinstruktion på ca 0.1 μs .

3.2.5 EXEKVERINGEN AV EN MASKININSTRUKTION

Vi har nu lärt känna alla blocken i det förenklade blockschemat, fig 3.44. Vi har också fått en ungefärlig inblick i hur styrenheten är uppbyggd. Det återstår alltså nu att se hela systemet i funktion när det utför ett program.

Vi har tidigare använt oss av demonstrationsprogrammet

LDA 4	B004
ADD 5	8805
STA 6	9806
HLT	0800

Låt oss nu se hur en av dessa maskininstruktioner exekveras av vår trafikcentral - dvs av styrenheten med det knepigt uppbyggda mikroprogramminnet. Låt oss välja LDA-instruktionen för vår studie!

Hanteringen av varje maskininstruktion sker alltid i två steg, hämta och verkställ (eller fetch och execute). Hämtningsrutinen är lika för alla instruktioner. Verkställandet eller exekveringen är givetvis olika för olika instruktioner.

3.2.5.1 Hämta instruktion

Vi antar nu att programräknaren har ställts in på den adress där första instruktionen är placerad. I vårt demonstrationsprogram ligger LDA-instruktionen i minnescell 0000. Vad händer när vi trycker på RUN-knappen?

Vi kan särskilja tre ofrånkomliga moment i hämtningsrutinen.

- Programräknarens adress måste läggas ut på adressbussen och minnet måste startas via kontrollbussen.
- När "färdig-signal" från minnet kommer in på kontrollbussen, skall data flyttas över från databussen till instruktionsregistret.
- Programräknaren skall inkrementeras så att den är redo att peka ut nästa instruktionsadress.

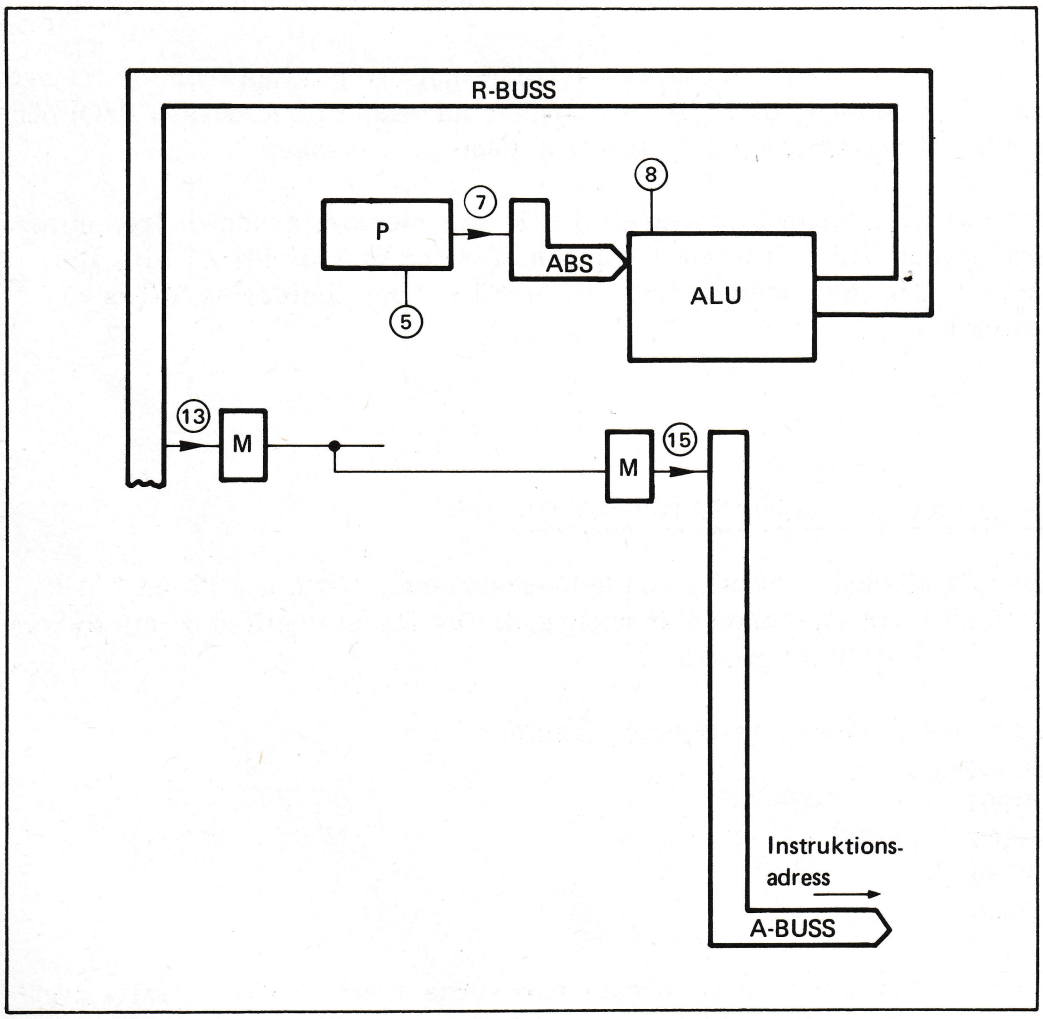


Fig 3.53 Utläggning av instruktionsadress på A-bussen (fig 3.44).
Kontrollsignaler: 5, 7, 8, 13, 15.

Dessa moment kan utföras på många sätt och en tänkbar lösning visas i fig 3.53 till 3.56.

I fig 3.53 läggs adressen från programräknaren P ut på adressbussen via kontrollerna 5, 7, 8, 13 och 15. Vi förutsätter att B-bussen innehåller talet 0.

I fig 3.54 kvarhåller M-registret (via drivsteget) adressen på adressbussen. Databussen riktas för mottagning (25) och en signalväg läggs till I-registret (22). Kontroll-logiken ger MST-signal (memory start) och samtidigt paussignal som stoppar klockan. Därmed står CPU:n och väntar på minnet, redo att ta emot instruktionen.

Pausen avbryts då MACK-signalen (memory acknowledge) från minnet inkommer på C-bussen. MACK-signalen betyder att data är tillgängliga på databussen och att I-registret därmed har laddats med den önskade instruktionen.

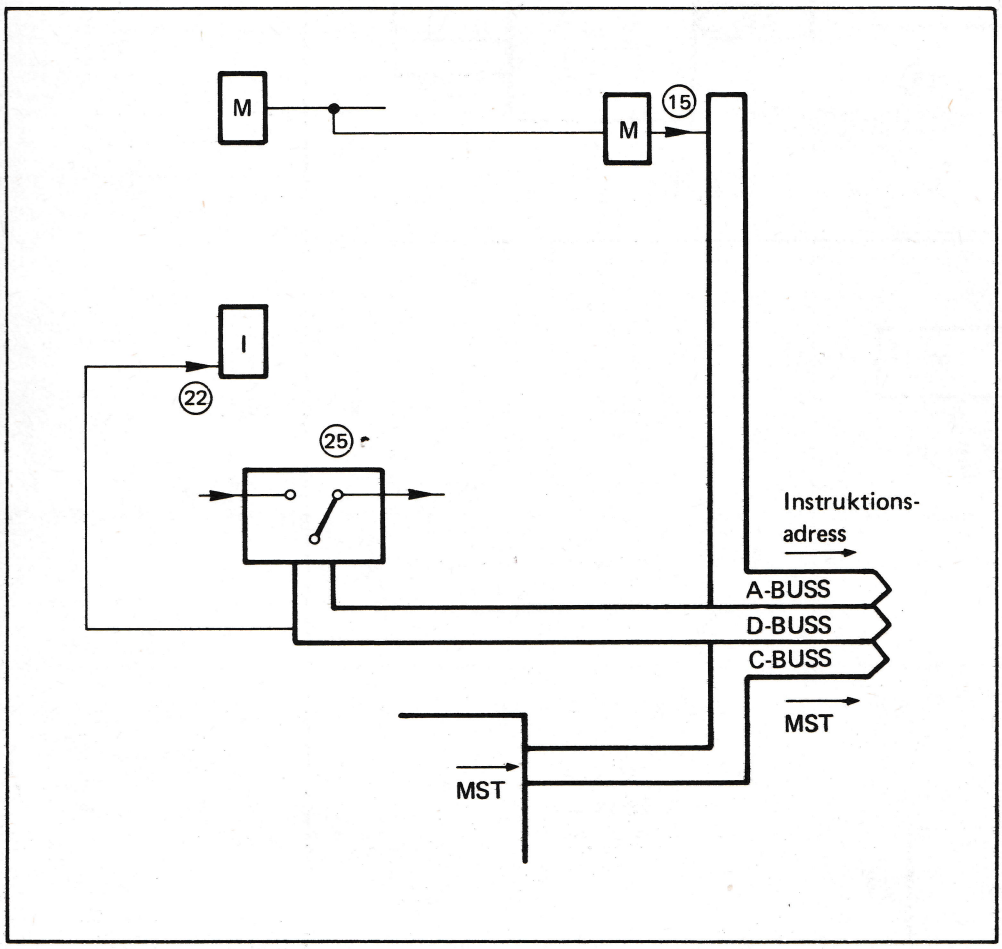


Fig 3.54 Väntan på minnet (15, 22, 25, MST, PAUS, MACK och PAUS SLUT).

I fig 3. 55 inkrementeras programräknaren (dvs P-registret). Det måste ske i två steg och här ser vi nu tydligt latchens uppgift.

Med 5, 7, 12 och 1 adderas talet ett till innehållet i P-registret och summan placeras i latchen L. P och L har alltså i detta skede olika innehåll och får därför ej kopplas samman. Fig 3. 55a.

Så bryts alla förbindningar varefter P sammankopplas med L. Det tidigare värdet i P som vi kan beteckna (P) förstörs nu och ersätts med (P) + 1.

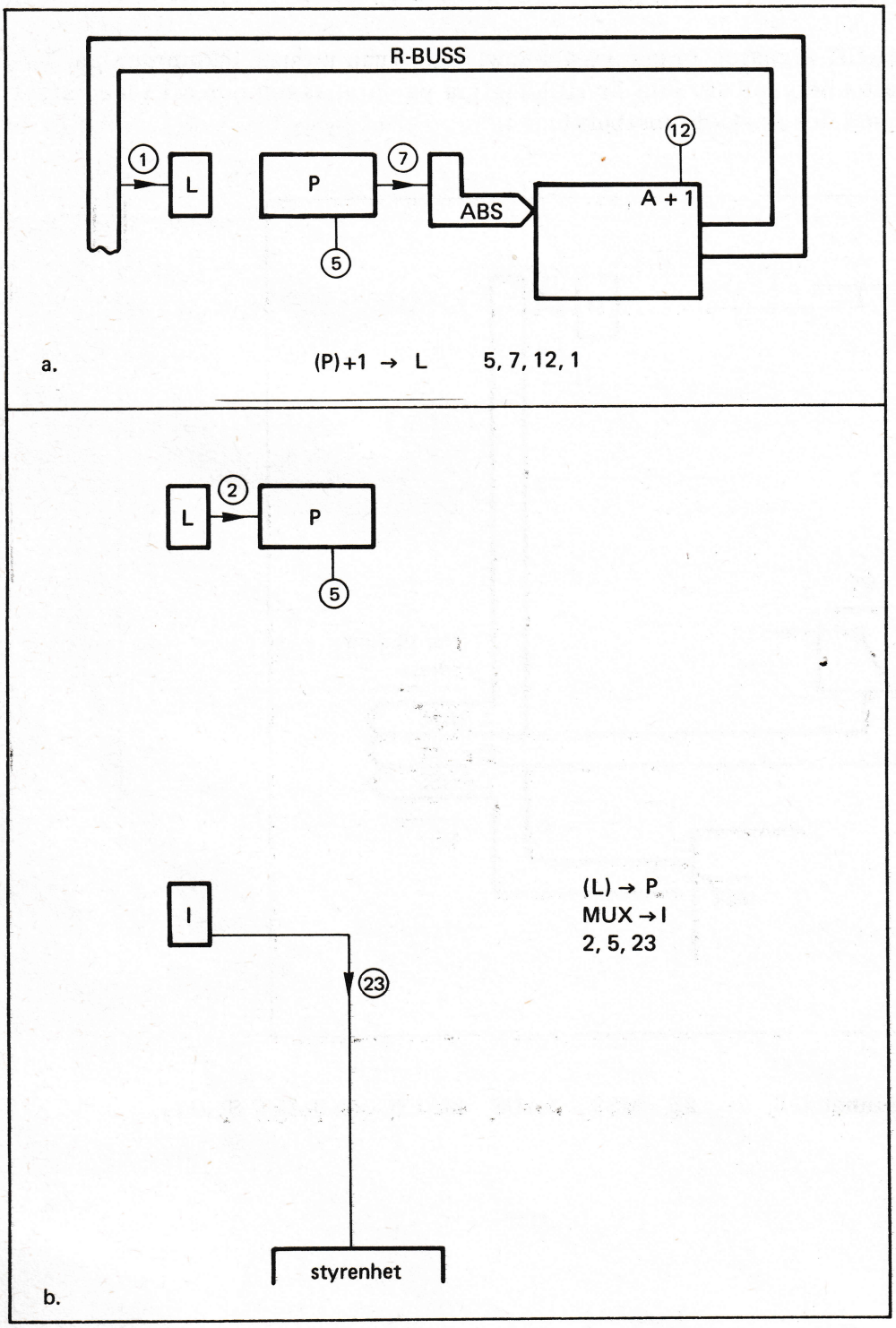


Fig 3. 55 Inkrementering av programräknaren.

Samtidigt överförs instruktionskoden som en adress via MUX:en till mikroprogramminnet. Vid efterföljande klockpuls hoppar därför programverket till adress

$B0_{16} = 176_{10}$

B0 är, som vi sett tidigare (i avsnitt 3.1.5.3), operationskoden för LDA uttryckt i hexadecimal form.

Därmed är hämtningsfasen slutförd.

Ett tänkbart utseende hos innehållet i mikroprogrammet framgår av fig 3.56. Siffrorna under rubriken "grindar" betyder alltså signal på respektive kontrollutgångar (0 - 25). Vissa utgångar går till kontroll-logiken. Denna kodning visas inte i detalj utan enbart med förkortningar under rubriken "kontroll". Längst till höger återfinns "nästa adress".

		Innehåll			
		Grindar	Kontroll	nästa adress	Fig
fetch	0	5, 7, 8, 13, 15		1	3.53
	1	15, 22, 25	MST, PAUS	2	3.54
	2	5, 7, 12, 1		3	3.55a
	3	2, 5, 23	MUX → I	4	3.55b
execute LDA	176	20, 8, 13, 15	MUX → N	177	3.58
	177	15, 25, 8, 1, 2, 3	MST, PAUS	0	3.59

Fig 3.56 Ett mikroprogram för maskininstruktioner LDA (i princip).

3.2.5.2 Verkställ LDA

En minnesrefererad instruktion (exempelvis LDA) innehåller som tidigare visats två väsentliga delar, en operationskod och en operandadress. Fig 3.57. Vi har i föregående avsnitt använt operationskoden för att finna adressen till LDA (dvs fack 176) i mikroprogramminnet. Eftersom MUX:en ställs i läge I (av mikroprogramcellen nr 3) överförs 176 som "nästa adress" till mikroprogramminnet (fig 3.51).

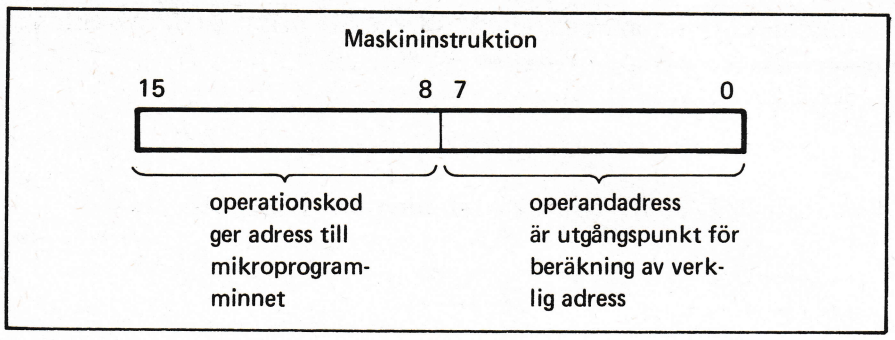


Fig 3. 57 Formatet för en minnesrefererad maskininstruktion.

När vi väl hamnat på denna adress fortsätter trafikdirigeringen från styrenheten på följande sätt:

- Operandadressen från I-registret läggs ut på adressbussen enligt fig 3. 58. Det är en ganska krokig väg - men vi skall strax förklara varför.
- D-bussen görs redo för mottagning och en signalväg läggs ut från D-bussen till ackumulatorn. Fig 3. 59. Minnet startas med kontrollsignalen MST på C-bussen.
- CPU:n väntar på svar (MACK) från minnet och när detta kommer är den önskade operanden redan inläst i ackumulatorn via den iordningställda signalvägen.
- Efterföljande klockpuls återför programverket till utgångspunkten - dvs till adress 0 för hämtning av nästa maskininstruktion (ADD 5) på den plats i minnet som utpekas av P-registret.

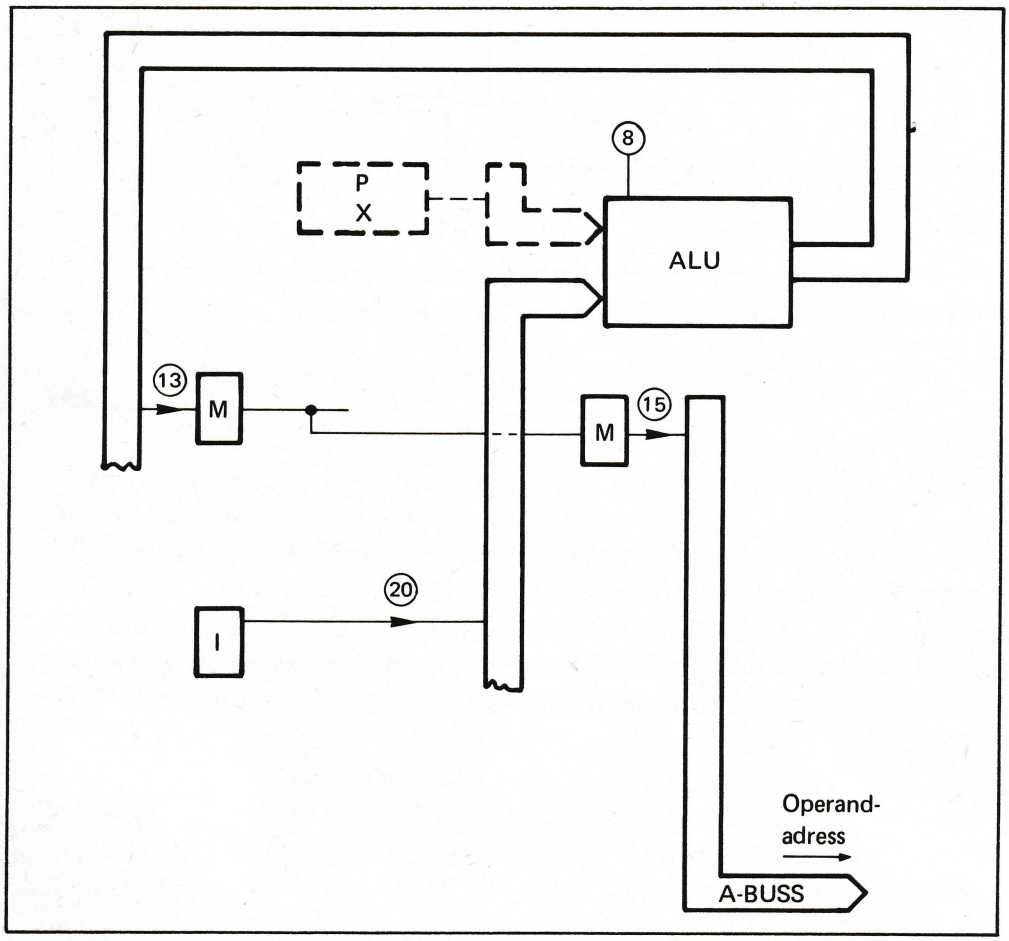


Fig 3. 58 Direkt adressering av minnet (20, 8, 13, 15).

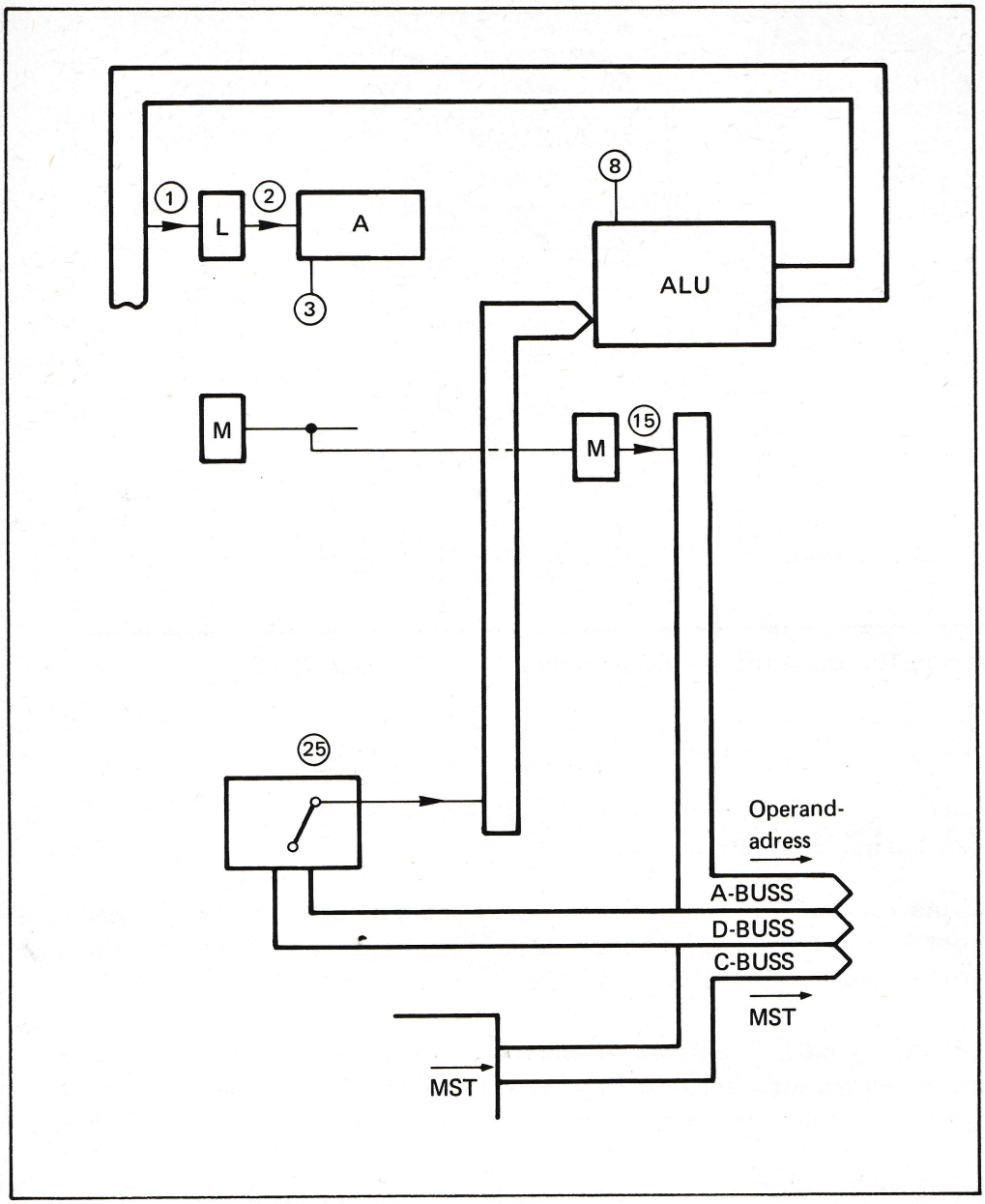


Fig 3. 59 Väntan på minnet (15, 25, 8, 1, 2, 3, MST, PAUS, MACK och PAUS SLUT).

Mikroprogrammet för exekveringen av LDA-instruktionen kan se ut som visas i fig 3. 56. (Givetvis är detta en grov förenkling av innehållet i ALPHA LSI:s programminne, men det ger oss en god uppfattning om principen för styrningen av signaltrafiken).

Tänk för övnings skull igenom och skriv upp tänkbara mikroprogram för maskininstruktionerna ADD, STA och HLT. Fig 3. 60 visar ett principförslag, men mikroprogrammen kan varieras på många sätt. I praktiken måste man ta hänsyn till en rad omständigheter som vi i detta sammanhang helt ignorerat.

Instruktionslistan för ALPHA LSI upptar som vi nämnt tidigare 168 maskininstruktioner. Alla dessa ligger lagrade som mikroprogram i de 14 kapslar som utgör mikroprogramminnet (fig 3. 50c).

ADD $88_{16} = 136$	136	20, 8, 13, 15	MUX → N	137	3.58
	137	15, 25, 8, 1, 3, 7	MST, PAUS	138	3.59
	138	2, 3		0	
STA $98_{16} = 152$	152	20, 8, 13, 15	MUX → N	153	3.57
	153	3, 7, 8, 24	MST, PAUS	0	
HLT $08_{16} = 8$	8		STOP	0	

Fig 3.60 Mikroprogram för maskininstruktionerna ADD, STA och HLT

3.2.6 ADRESSERING AV EN OPERAND

En minnesrefererad instruktion består av 8 bitar operationskod och 8 bitar operandadress. Man frågar sig hur det är möjligt att addressera alla fack i minnet med enbart åtta bitar. Maximala antalet kombinationer av åtta binära siffror är ju $2^8 = 256$.

Om vi använder operandadressen direkt kan vi enbart adressera de 256 första orden i minnet. Denna minnesarea brukar därför kallas "scratchpad" (kladdpapper, block för tillfälliga anteckningar). Vårt demonstrationsprogram arbetar enbart inom scratchpad.

För att kunna adressera fack utanför scratchpad måste vi tillgripa något knep. Det är för detta ändamål vi ritat in P- och X-registren i fig 3.58. Det vanliga knepet är att den verkliga adressen beräknas med hjälp av operandadressen i instruktionsregistret samt P- eller X-registret. Det kostar ju inget extra i fig 3.58 att addera adressen i I-registret med innehållet i X- eller P-registret innan den slutliga adressen placeras i M-registret (och därifrån på adressbussen).

När man adderar innehållet i X-registret brukar man tala om indexerad adressering. Den används speciellt när man vill adressera en löpande följd av minnesfack. Det sker enkelt genom att man kan inkrementera X-registret för varje ny adressering.

Adderar man P-registret kallas adresseringen relativ - dvs relativ till P. Man kan härvid både addera till P eller subtrahera från P och kallar detta "relative to P forward" och "relative to P backward". Relativ adressering är den vanligaste adresseringsmoden.

Man kan också låta instruktionens operandadress peka ut det fack i minnet där den verkliga adressen är lagrad. Man kallar detta indirekt adressering och operandadressen är i detta fall en "pekare" (address pointer).

Hur vet då styrenheten vilken adress den skall lägga ut på adressbussen? Det ligger kodat i operationskoden så som visas i fig 3.61.

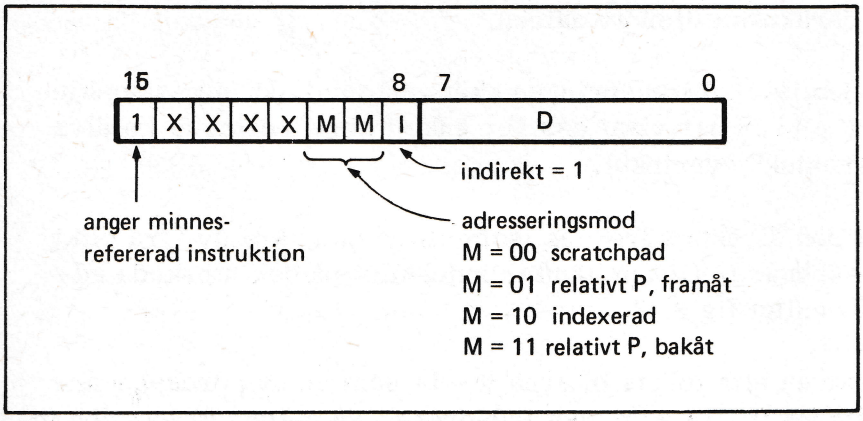


Fig 3.61 Olika adresseringsmoder hos minnesrefererad instruktion

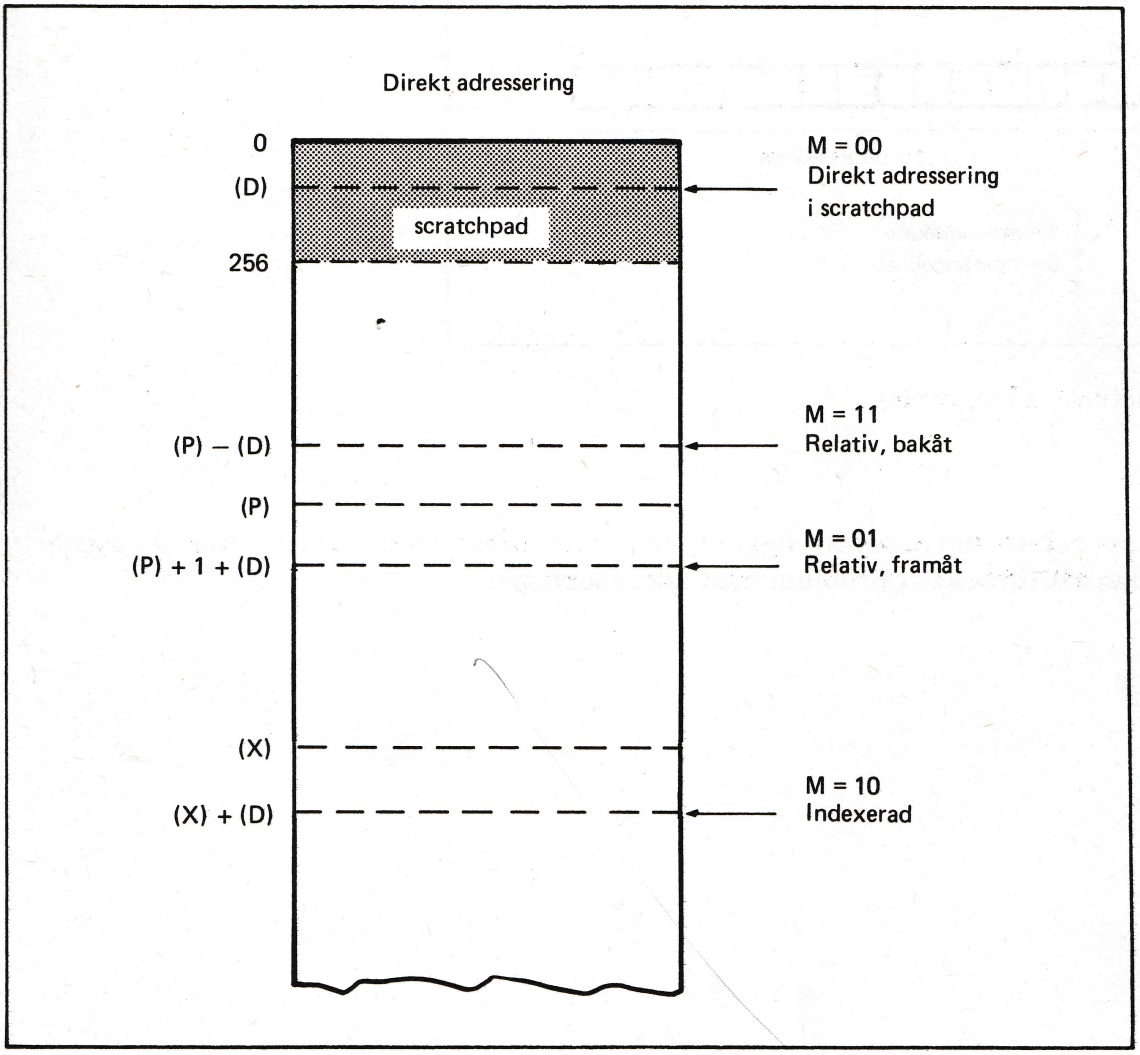


Fig 3.62 Olika moder för direkt adressering
(D) = innehåll i I-registrets datadel (operandadress)
(P) = innehåll i P-registret
(X) = innehåll i X-registret

Låt oss beteckna den del av instruktionsregistret som innehåller operandadressen för D och dess innehåll för (D). De olika adresseringsmoderna ger oss då de adresser som framgår av fig 3. 62. De tre adresser som inte ligger i scratchpad (dvs relativ bakåt, relativ framåt och indexerad) beräknas alltså av datorn.

Vi ska inte här fördjupa oss i detaljer. Adresseringen sköts automatiskt vid assembleringen som vi snart skall se. Fig 3. 58 har visat oss hur enkelt adresseringen i själva verket tillgår, sett från "elektronisk" synvinkel.

Vid indirekt adressering utgör den direkta adressen (som alltså kan vara av fyra olika moder enligt fig 3. 62) en adresspekare. CPU:n hämtar innehållet på den utpekade adressen och tolkar detta innehåll enligt fig 3. 63.

Om bit 15 i det hämtade ordet är en etta tolkas bitarna 0 - 14 som en ny adresspekare. Om bit 15 är en nolla tolkas bitarna 0 - 14 som operandadress. På detta sätt kan den indirekta adresseringen ske i flera steg.

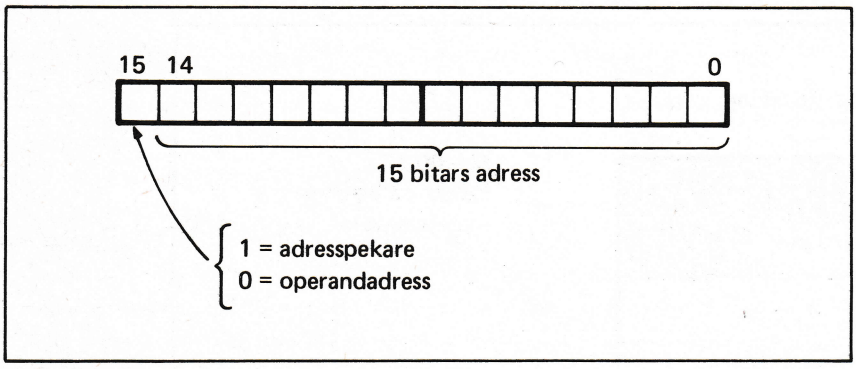


Fig 3. 63 Indirekt adressering.

Fördelen med relativ adressering ligger i att programmet enkelt kan relokeras (flyttas i minnet) utan att förorsaka problem med adresseringen.

3.2.7 HOPP

Vi har tidigare sett att programhopp (jump) är helt nödvändiga om man skall kunna göra program som tar hänsyn till delresultaten av datorns beräkningar.

Hoppinstruktionerna har (liksom de tidigare beskrivna minnesrefererade instruktionerna) en adressdel som anger vart hoppet skall ske och en operationskod som anger under vilka betingelser hopp skall äga rum.

I ALPHA LSI omfattar hoppinstruktionens adressdel 6 bitar och bit nr 6 anger i vilken riktning relativt P hoppet skall ske. Fig 3.64. Ett hopp kan därför ske maximalt 63 steg bakåt relativt P eller 64 steg framåt.

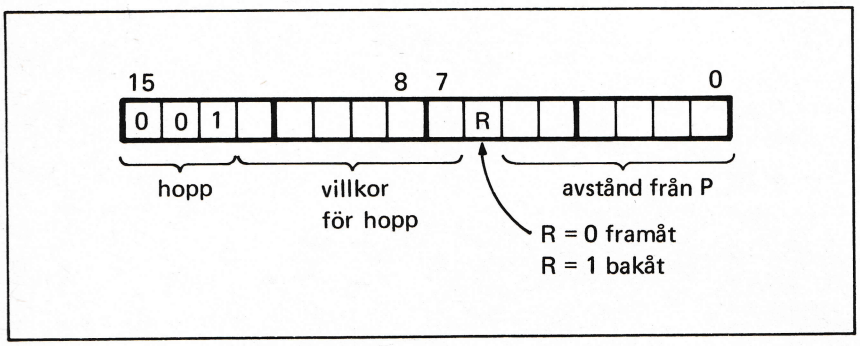


Fig 3.64 Format för hoppinstruktion.

Sex bitar av instruktionen används för hoppvillkoren, det blir inte mindre än 64 olika villkor. Dessa villkor testas av styrenheten genom att den med ALU:ns hjälp tar reda på om innehållet i A- eller X-registren är = 0, större än 0 eller mindre än 0. Styrenheten kan även testa eventuellt spill (OV = overflow) från ALU:n, dvs om dess innehåll överskrider ALU:ns kapacitet. Även SENSE-registret kan testas som hoppvillkor.

Det finns även ett hopp utan villkor (JMP) och här sker adresseringen av hoppet på samma sätt som för minnesrefererad instruktion (dvs som i fig 3.61).

För att datorn skall kunna utföra subrutiner och sedan hitta tillbaks till det ställe i programmet som den tidigare lämnat, så finns en "hopp och lagra"-instruktion (JST = jump and store). När denna exekveras lagras programräknarens innehåll på hoppadressen (som första steg i subrutinen).

En dator med väl utvecklade hoppinstruktioner är kraftfull både för beräkningsuppgifter och vid hantering av data.

Vi har i fig 3.44 ej utritat spillvippan (OV), sensevippan (SS) eller utgångar från ALU:n för avkänning av innehåll. Vi skall inte heller beskriva några mikroprogram för dessa hoppinstruktioner. De skiljer sig inte i princip från de mikroprogram vi tidigare beskrivit.

3.2.8 KRAFTFÖRSÖRJNINGEN

Elektroniska kretsar drar effekt och det medför två problem för datorkonstruktören. Först måste han konstruera ett nätaggregat (power supply) som avger den önskade effekten med noggrant reglerad spänning. TTL-kretsarna i ALPHA LSI kräver 5 V matningsspänning och enbart CPU-kortet drar 6,5 A.

Den tillförda elektriska effekten övergår till värmeförluster i kretsarna. Bara CPU-kortet bidrar alltså med 30 W för "uppvärmningen" av minidatorn. Eftersom tillförlitligheten hos elektroniska kretsar avtar vid högre temperatur, måste värmeeffekten kylas bort.

Fig 3.65 visar fläktarna som blåser ut energin när den har fullgjort sin uppgift - dvs försett kretsarna med nödvändig matning.

Fig 3.65 Fläkten för kylning.

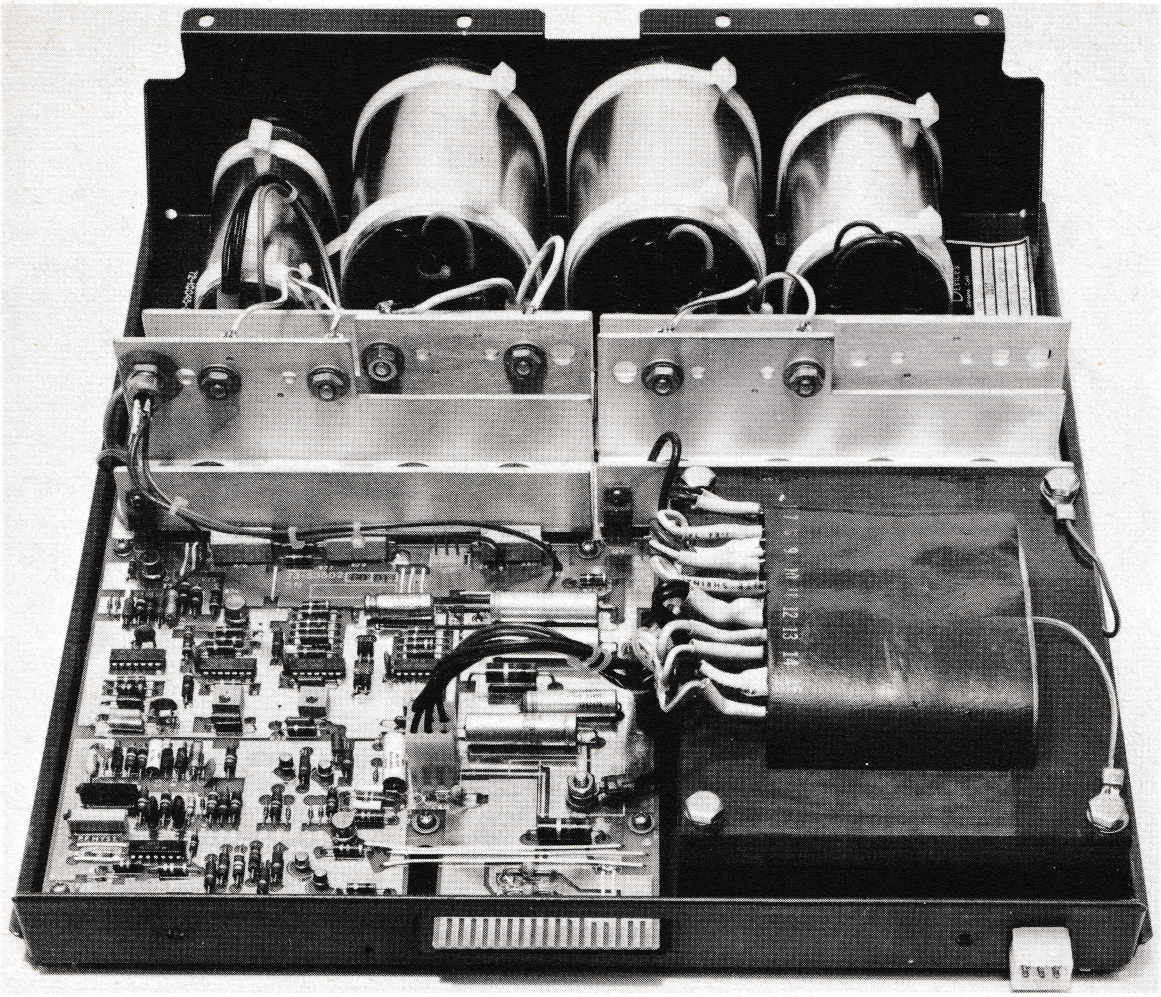


Fig 3.66 Kraftförsörjningen (power supply) (Observera de stora kondensatorerna som håller kvar matningsspänningen 10 ms vid nätavbrott).

Fig 3.66 visar ALPHA LSI:s nätdel. Den ger tillräcklig effekt för att mata både CPU:n och ett antal minnes- och I/O-enheter.

Nätdelen avger tre spänningar och får maximalt belastas med

28 A på +5 V-utgången
9 A på 12 V-utgången
3 A på -12 V-utgången.

Ytterligare en 5 V-spänning finns tillgänglig. Den kallas 5 H (hang) och har betydelse vid strömavbrott.

Om nätspänningen faller ifrån får CPU:n en avbrottssignal (power fail) av en krets som känner nätspänningen. Härvid sker ett avbrott av samma typ som vi tidigare beskrivit i avsnittet om ALPHA LSI:s styrenhet. Pågående program avbrytes och samtliga register lagras undan i minnet.

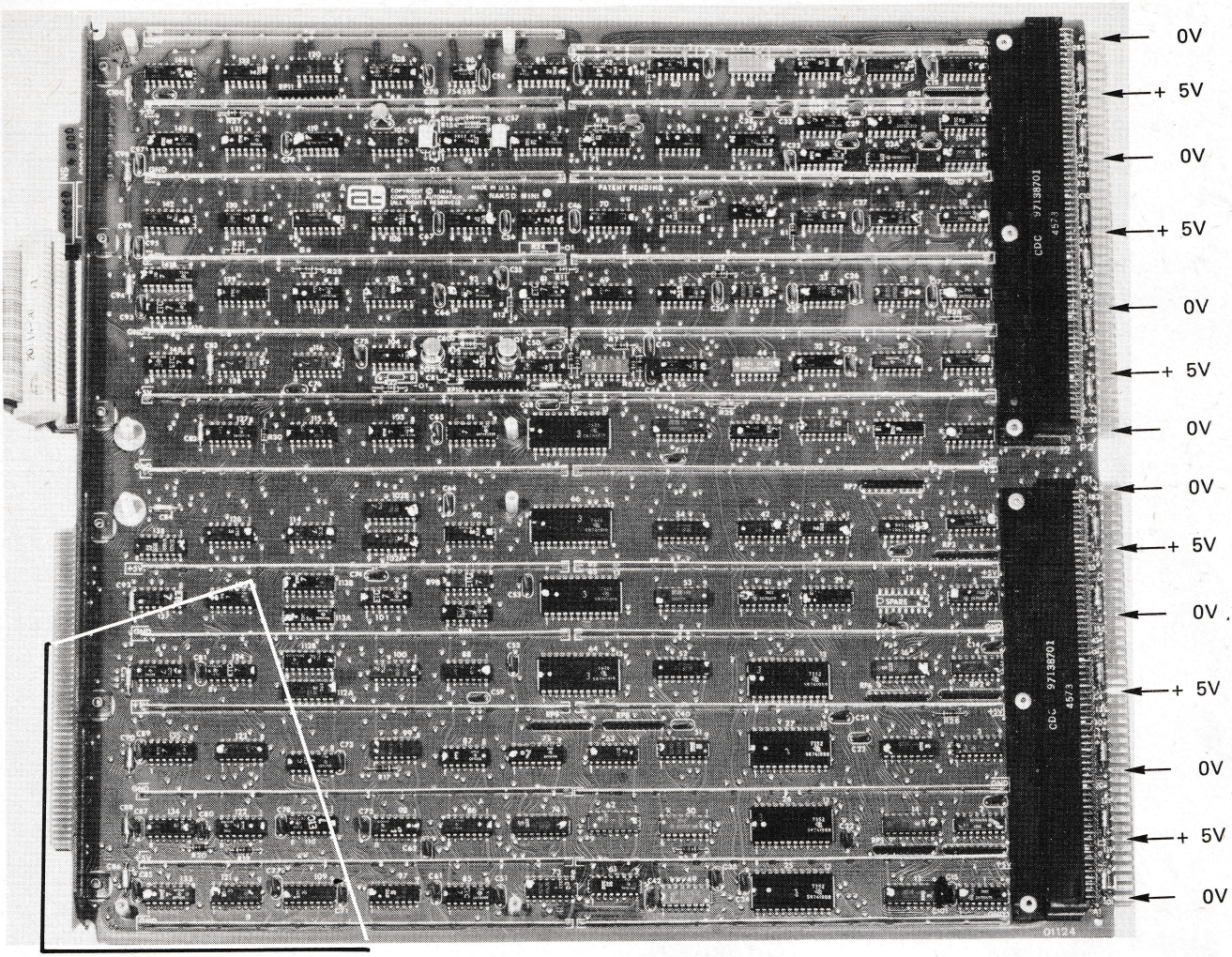
Det tar minst 10 ms innan matningsspänningarna sjunkit så mycket att CPU:n ej längre kan fungera och under denna tid hinner CPU:n gott och väl att lagra undan alla registerdata. När de tre matningsspänningarna försvinner ligger hangspänningen (5H) kvar ytterligare någon sekund och förhindrar att minnet får falska signaler vid spänningsbortfallet.

När nätspänningen återkommer (efter strömavbrottet) inkopplas först hang-spänningen och därefter övriga matningsspänningar. När dessa stabiliserats ger "power-fail"-kretsen CPU:n adressen till en igångkörningsrutin. Härvid läses tidigare undanlagrade data tillbaks till CPU:ns register varefter arbetet återupptas där det avbröts vid spänningsbortfallet. Det behövs alltså en hel del logikkretsar även i kraftförsörjningen!

Hur ser då strömvägarna för matningsspänningen ut på CPU:n och på övriga kretskort?

Vi har tidigare enbart behandlat signalvägarna på kretskorten. För att få fram matningsspänningen 5 V till samtliga kretsar fordras bussar även för likströmmen. Det är härvid ett avsevärt problem att förhindra att störningar utbreder sig som signaler överlagrade på matningsspänningen.

Fig 3.67 visar hur man lagt jord (GND = ground) och + 5 V på 13 st parallellt löpande skenor över CPU-kortet. Matningsspänningen silas av elektrolytkondensatorer i båda ändar av skenorna. Vid varje krets sitter dessutom en snabb keramikkondensator parallellt med matningsspänningen. Dess uppgift är att sila bort "strömspikar".



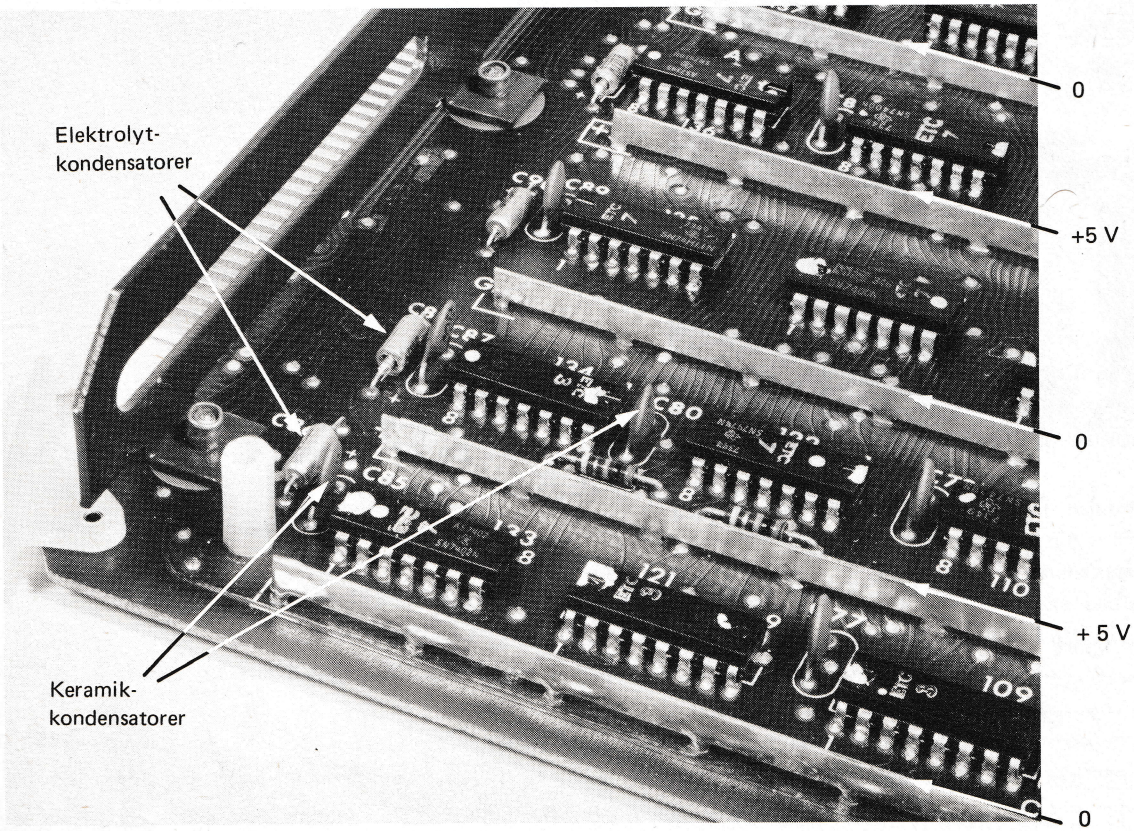


Fig 3. 67 Matningsspänningen på CPU-kortet.

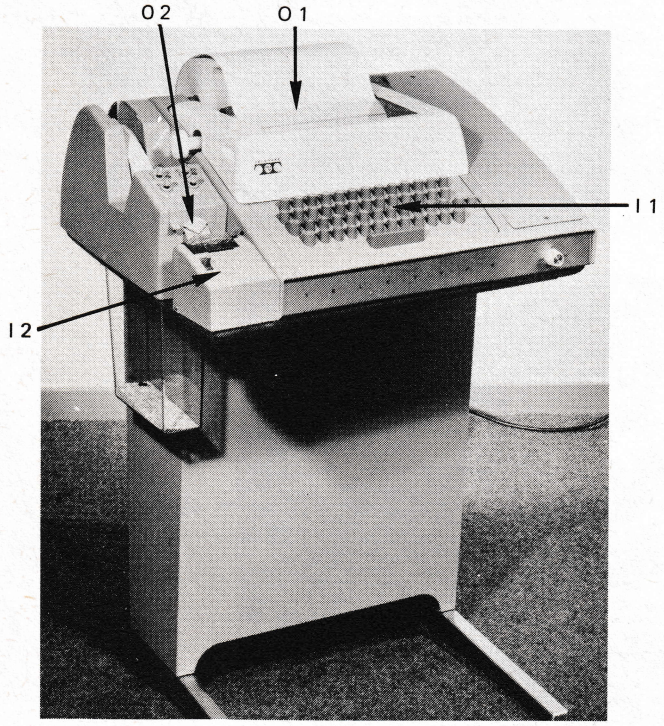
3.3 Input/output

En dator måste kunna meddela sig med omvärlden. I ALPHA LSI sker detta via Maxibussen. Via Maxibussen kan olika I/O-enheter adresseras, data kan sändas till eller från datorn och styrorder kan utgå från datorn till omgivningen eller vice versa.

Vi ska i detta avsnitt bekanta oss med en speciell I/O-enhet, TTY:n (eller fjärrskrivmaskinen). Vi ska se vilka funktioner den kan fullgöra och hur dessa kan styras under programkontroll.

3.3.1 TTY:N OCH DESS FUNKTIONER

De flesta minidatorer har en TTY för kommunikation med operatören. Har man lyft en TTY undrar man varför en TTY måste väga så mycket. Lyfter man på plastkåpan och ser mängden med mekaniska stänger, hjul, kopplingar etc så undrar man varför allt



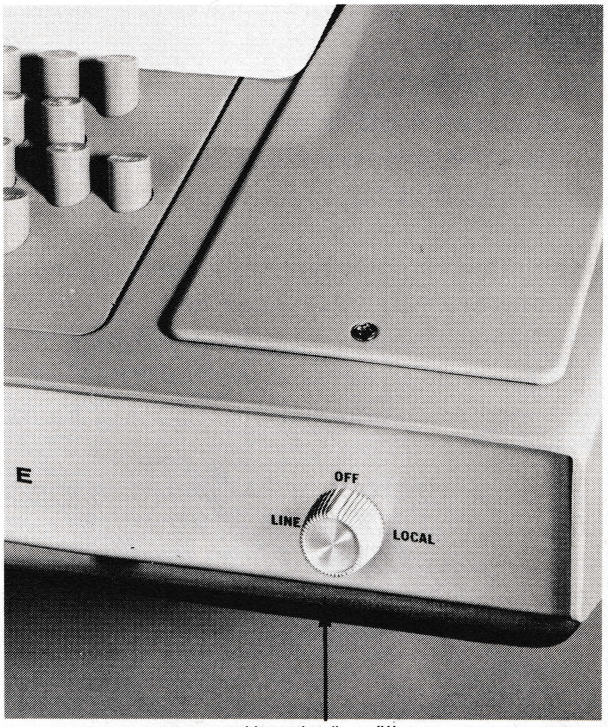
IN (till datorn)		UT (från datorn)	
I 1	Tangentbord (keyboard)	O 1	Teckenskrivare (typing unit)
I 2	Remsläsare (tape reader)	O 2	Remsstans (tape punch)

Fig 3.68 TTY av typ ASR 33.

detta är mekaniskt och inte elektroniskt. Vi har ju gått ifrån mekaniska räknemaskiner till elektroniska räknedosor. Och visst finns det elektroniska motsvarigheter!

Men när man ska betala priset för en remstans, en remläsare, en teckenskrivare och ett tangentbord - då finner man att TTY:n fortfarande utgör ett både ekonomiskt och slitstarkt alternativ, trots vikten och mekaniken. TTY:ns fyra funktioner framgår av fig 3. 68.

Låt oss pröva funktionerna! Huvudströmställaren sitter framtill och till höger, fig 3. 69b. Ställ den i läge LOCAL. Om nätkabeln är ansluten skall motorn starta och det är klart att trycka på tangenterna.



Huvudströmställare

Fig 3. 69 TTY:n i närbild.
a. Remsstans och remsläsare
b. Huvudströmbrytare

Varje tangentnedtryckning ger oss motsvarande utskrift på teckenskrivarens papper. Tydligen är tangentbordet inkopplat till teckenskrivaren precis som i en vanlig skrivmaskin (med kula). Observera att nollan skrivs med snedstreck (Ø) eller "slashed" för att man skall kunna särskilja nollan från bokstaven O. Detta skrivsätt kommer vi att använda i fortsättningen.

Vi trycker nu in knappen märkt ON på remsstansen fig 3. 69a. Nu visar det sig att vi får två "utskrifter" när vi i fortsättningen trycker ned olika tangenter. Dels får vi (liksom tidigare) utskriften på papperet och dels får vi en hålremsa.

I själva verket är hålstansen mekaniskt kopplad till teckenskrivaren och stansar därför allt som skrivs när ON-knappen är intryckt. Vi kan enkelt få reda på den kod som hålstansen använder genom att jämföra pappersutskriften med hålremsan, fig 3. 70.

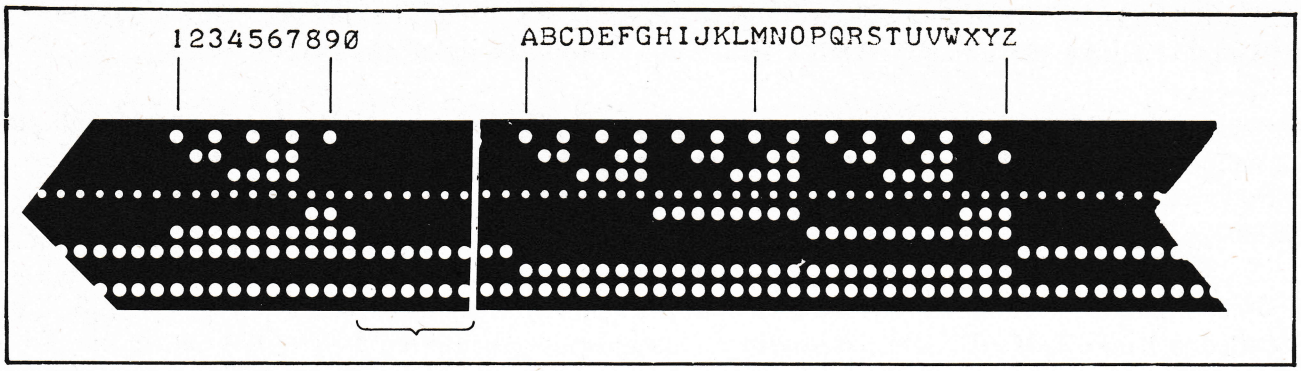


Fig 3. 70 Hålremsans information.

Låt oss nu flytta över hålremsan till remsläsaren. Vi startar remsläsaren genom att ställa omkopplaren START-STOP-FREE i läge START (fig 3. 69a). Remsan stegas fram och samtidigt får vi nu en ny utskrift av allt som vi tidigare stansat in på remsan.

Remsläsaren och tangentbordet avger elektriska signaler i form av spänningar på åtta ledningar. När vi trycker ned en tangent startas en omkopplare (distributor) som avsöker dessa åtta ledningar i tur och ordning. På omkopplarens axel sitter åtta kamhjul som via mekaniska hävstänger ställer in åtta kodstänger (codebars) under tryckverket (i enlighet med de inkommande elektriska signalerna).

Själva tryckverket är en liten vagn som stegas fram av en rem från vänster till höger (fig 3. 73). På denna vagn sitter typhuvudet som styrs av kodstängerna. Kodstängerna nr 1 - 4 vrider in typhuvudet (som utformats som en cylinder) i lämplig vinkel. Kodstängerna 5 och 7 ställer in typhuvudet i lämplig höjd. En liten gummiklubba "slår" sedan typhuvudet mot papperet över tryckvalsen. Med fyra positioner i höjdled och 16 positioner i vinkel får vi 64 tecken. Typhuvudet är avbildat på fig 3. 71.

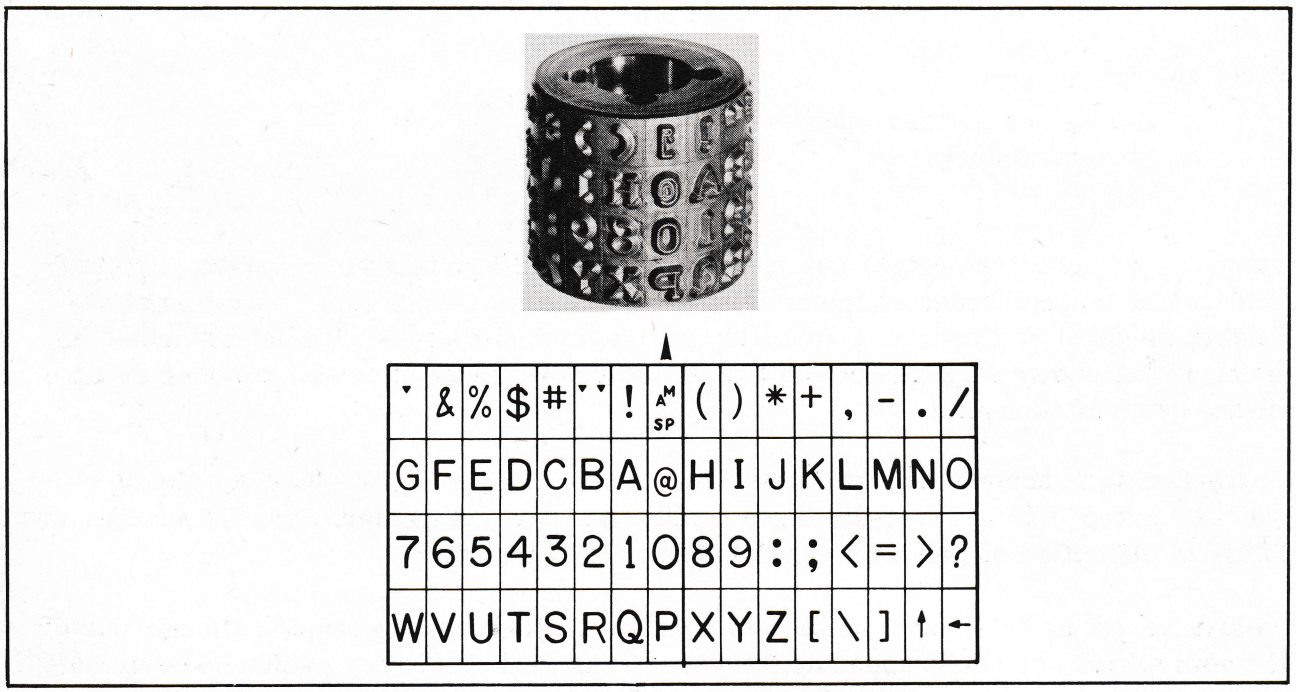


Fig 3. 71 Tryckcylindern i en TTY.

Mekaniken i en TTY är fantastisk. Man ger den inte rättvisa genom att beskriva den med ord - den måste ses! Egentligen är det synd att all sådan mekanik sakta men säkert försvinner och ersätts med elektroniska kodare.

Även tangentbordet i en TTY är ett mekaniskt underverk av hävstänger. Vid nedtryckning av en tangent trycker en skena ner en kombination av åtta längsgående kodstänger (codebars). Vid tangentbordets högra sida sitter åtta kontakter som ger utsignal i enlighet med den nedtryckta kombinationen.

Fig 3. 72 visar tangentbordet och man ser i högra kanten såväl de längsgående stängerna som kontaktfjädrarna.

Fig 3. 73 visar tryckverket (med färgband och typhjul) som glider fram över kodstängerna.

Hålstansen är mekaniskt kopplad till kodstängerna under tryckverket. Här överförs kodstängernas inställning till åtta stansar. Dessa stansar hål på pappersremsan i enlighet med kodstängernas inställning. Hålstansen innehåller även en stegmekanism för frammatning av remsan. Fig 3. 74.

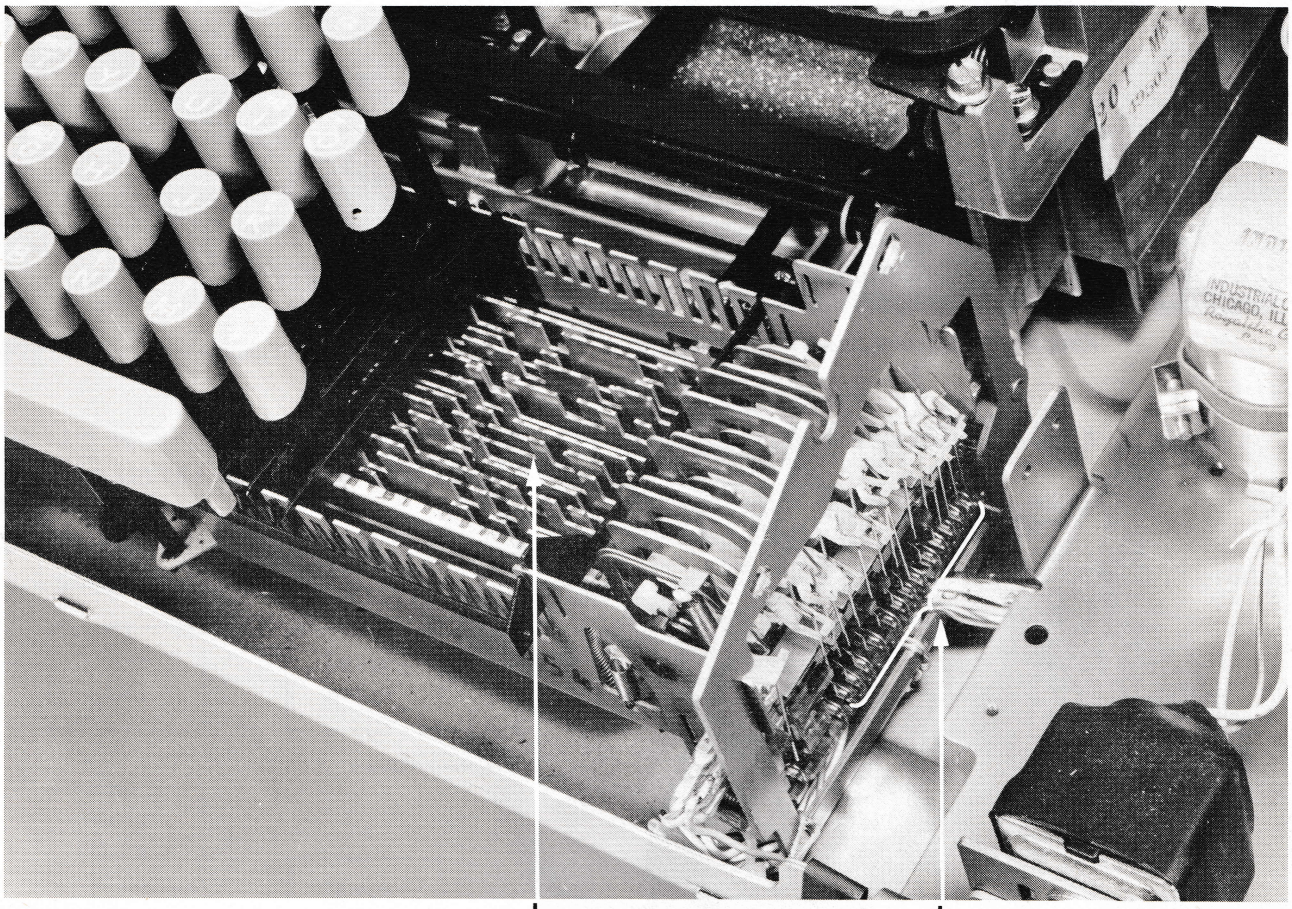


Fig 3. 72 Tangentbordet och dess mekanik.

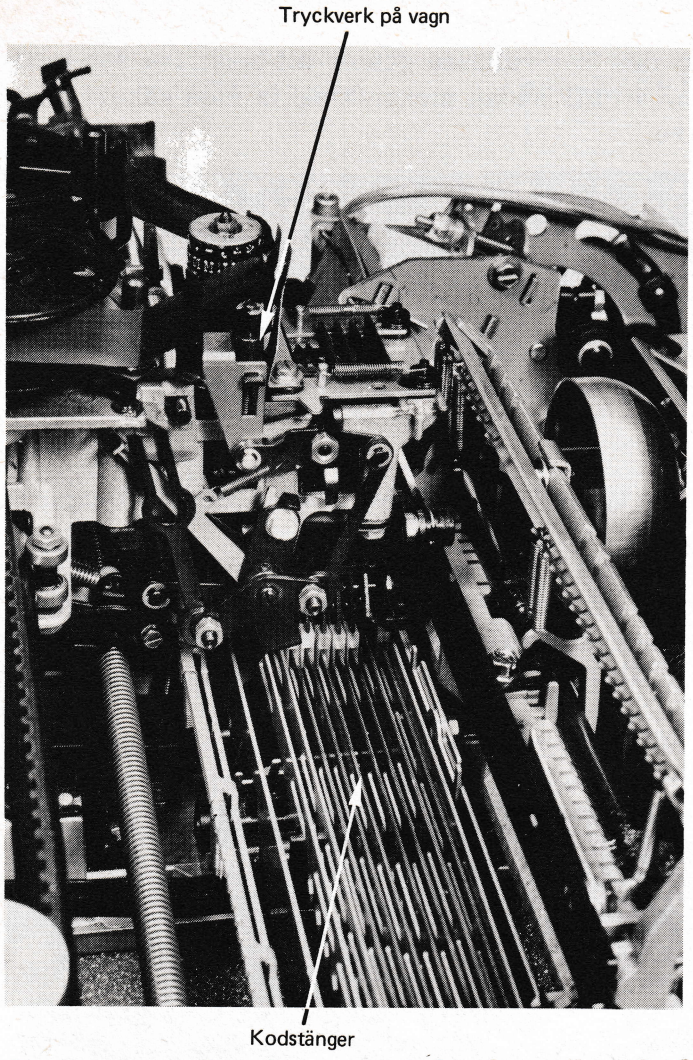


Fig 3. 73 Tryckverket som glider fram över kodstängerna.

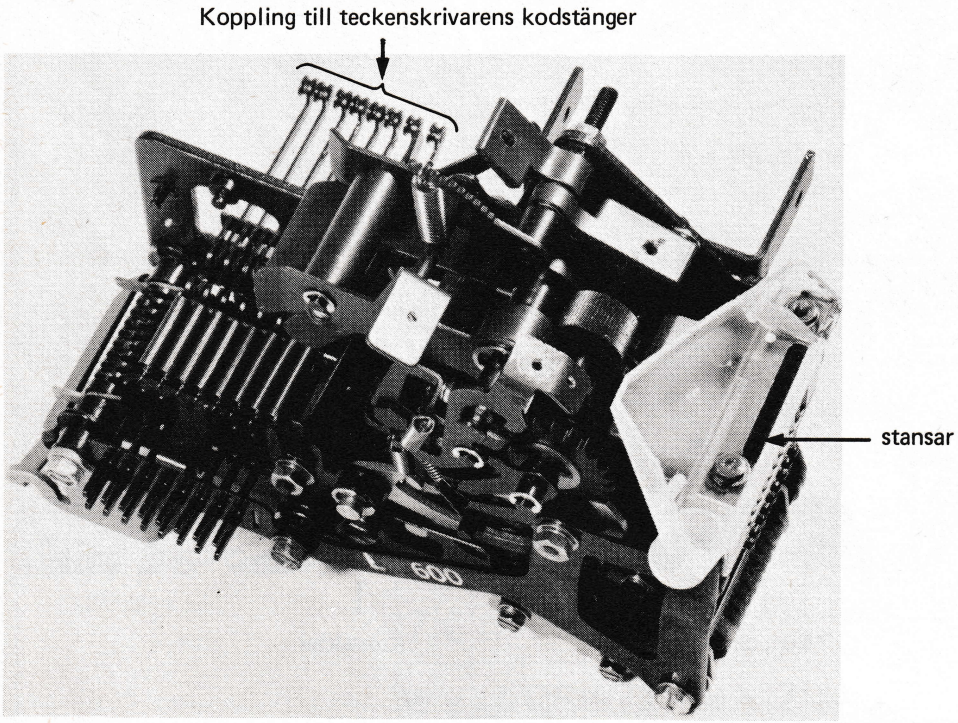
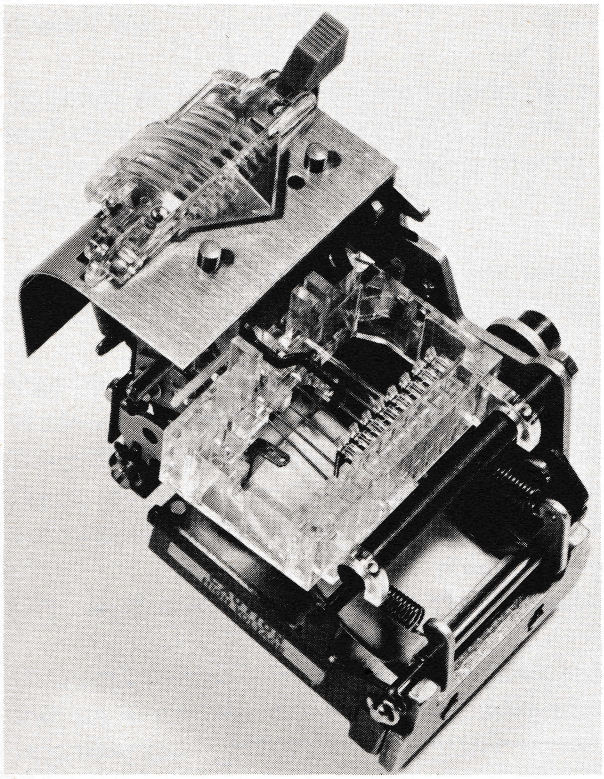


Fig 3. 74 Hålstans.

Remsläsaren är enklare till sin uppbyggnad. Även remsläsaren arbetar rent mekaniskt och skjuter fram åtta "känselstänger" mot hålremsan. Om remsan har hål kommer motsvarande känselstång att skjutas upp och denna rörelse ger en elektrisk kontakt. Signaler från läsarens åtta kontakter förmedlas via åtta ledningar till omkopplaren (distributor) precis som de åtta signalerna från tangentbordets kodstänger. Fig 3. 75 visar en närbild av remsläsaren och dess manöverorgan. Omkopplaren har som synes tre lägen, START, STOP och FREE.



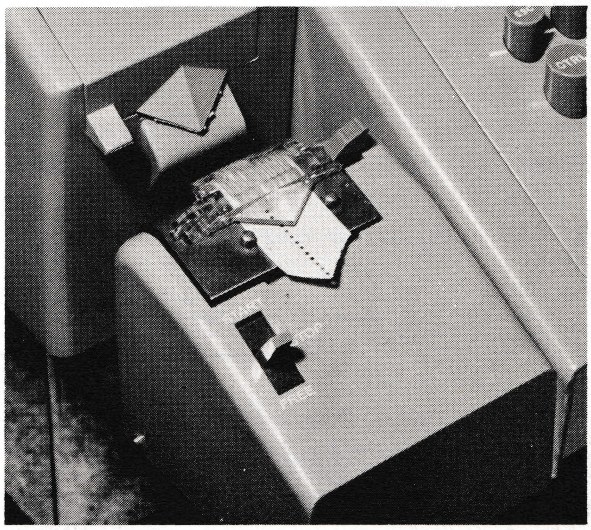


Fig 3. 75 Remsläsaren.

De åtta ledningarna från remsläsare och tangentbord är parallellkopplade. Man får därför inte trycka på tangenterna när läsaren är i funktion.

TTY:n har alltså fyra funktioner och i fig 3. 76 har vi tecknat ett blockschema.

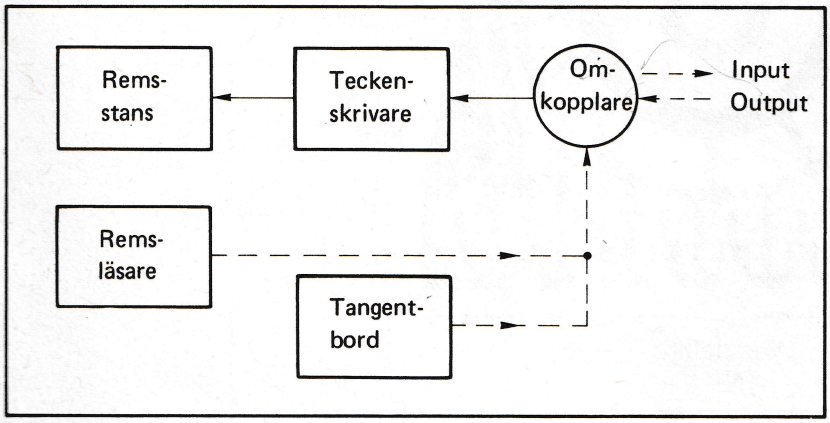


Fig 3. 76 Blockschema för TTY

——◄—— mekanisk överföring

– – ► – – elektrisk överföring

3.3.2 TTY:N SOM SÄNDARE OCH MOTTAGARE

TTY:n konstruerades ursprungligen för överföring av skriftliga meddelanden per telegrafledning, fig 3.77a. Ledningar är dyra och informationen måste då sändas i serieform. Det är här TTY:ns omkopplare (distributor) kommer in i bilden. Fig 3.77b.

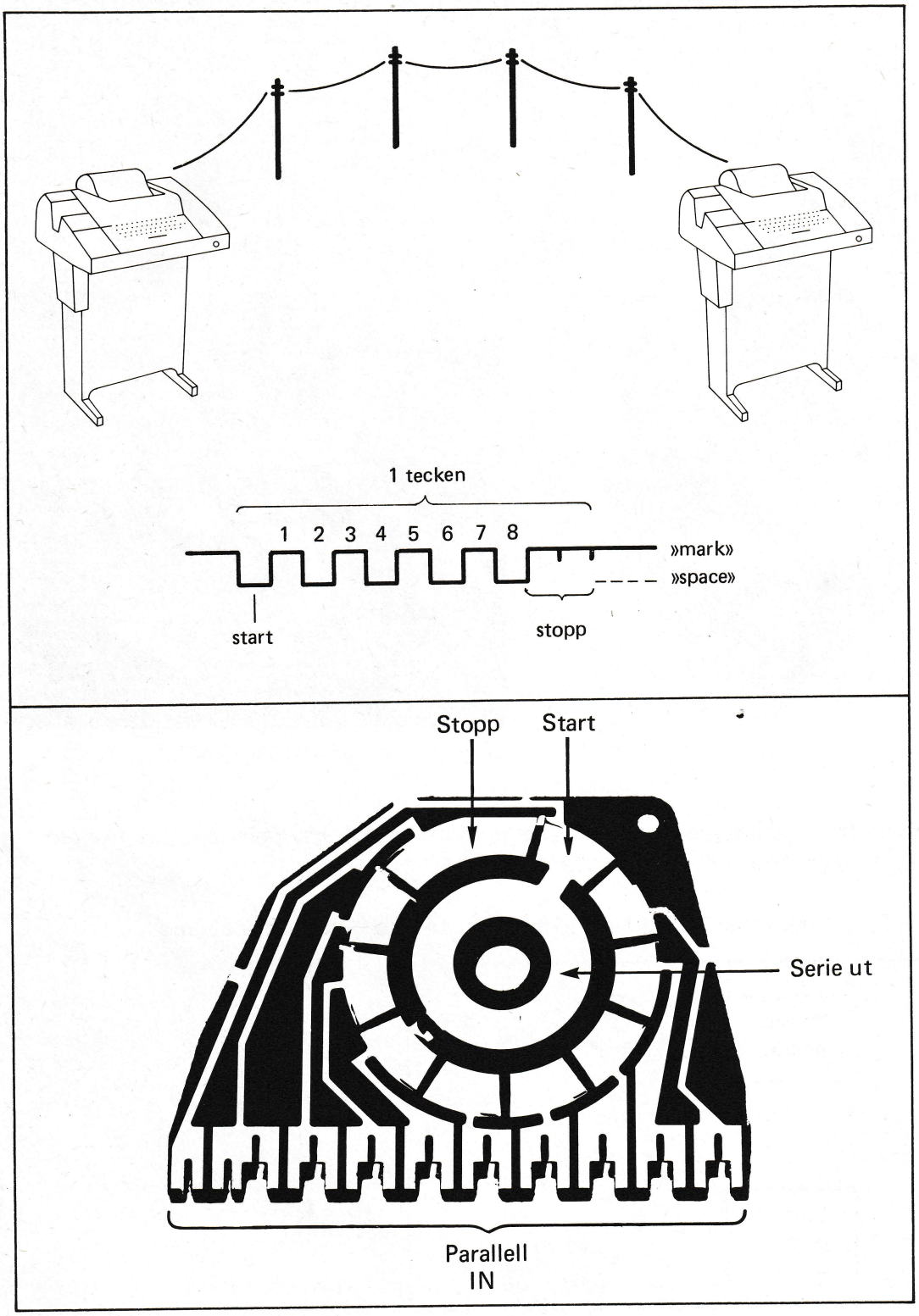


Fig 3.77 a. TTY:ns tecken - en serie om 11 pulser
b. Parallell-serieomvandlaren i en TTY.

När vi på sändarsidan trycker ned en tangent avger tangentbordet en kombination av 8 signalspänningar. Samtidigt sätts omkopplaren i rörelse. Den drivs med konstant hastighet från en motor och avsöker i tur och ordning 11 kontakter. Vi får därför en utsignal som uppdelas i 11 tidsmoment, vardera 9,09 ms långt.

Utgången arbetar vanligen med strömslinga (current loop) och signalen består därvid antingen av 20 mA ström (mark) eller strömavbrott (space). Finessen med en strömslinga ligger i att strömmen i mottagaren blir konstant oavsett vilken resistans ledningen har.

Normalt är ledningen strömförande (mark) och ett tecken börjar därför alltid med en strömlös "startbit" (space) på 9,09 ms. Därefter följer 8 tidsmoment som ger en kombination av "ström" och "strömavbrott". Tecknet avslutas med två "stoppbitar" som alltid är ström (mark).

Man överför på detta sätt 11 bitars tecken med en hastighet av 10 tecken per sekund eller 110 bitar per sekund. Överföringshastigheten kallas vanligen 110 baud.

Vad händer då på mottagarsidan? Startbiten startar mottagarens omkopplare som därvid fördelar efterföljande 8 pulser på omkopplarens olika kontakter (därav namnet distributor). Härvid ställs kodstängerna under tryckverket i en kombination som motsvarar det mottagna tecknet. När första stoppbiten kommer slår typhuvudet mot pappersrullen med höjd och vinkel inställda i enlighet med kodstängernas lägen.

Om stansen är inkopplad får vi det mottagna meddelandet även i form av en hålremsa.

Omkopplaren i TTY:n fungerar alltså som en serie-parallellomvandlare där serieinformation körs över den yttre ledningen medan TTY:n internt arbetar med 8 bitars parallellinformation.

3.3.3 ASCII

Vid överföring av meddelanden används vanligen American Standard Code for Information Interchange, eller ASCII-kod som den ofta kallas.

ASCII är uppbyggd av 7 informationsbärande bitar och en paritetsbit. Paritetsbiten, dvs bit nr 8, är tillagd som en kontrollbit (parity check). Om bit nr 8 inte utnyttjas för paritetskontroll är den alltid "mark".

Om paritetsbiten utnyttjas använder man jämn paritet. Det betyder att bit 8 skall vara sådan, att man får totala antalet markeringar (marks) jämnt (när man räknat samman antalet "marks" för samtliga åtta bitar).

ASCII-koden innehåller därmed $2^7 = 128$ tecken, men av dessa trycker vår TTY enbart 64. TTY:n använder exempelvis enbart versaler ("stora bokstäver") och inga gemener. 32 av tecknen är kontrollsignaler som inte ger utskrift på TTY:n.

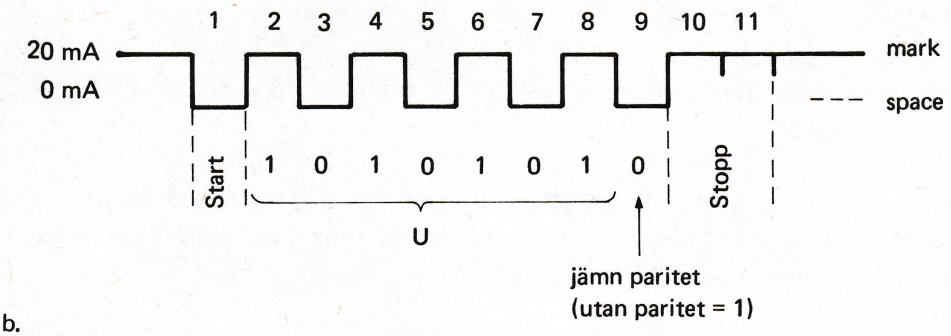
Fig 3. 78a visar några vanliga tecken ur ASCII-koden. Tecknet för U (med jämn paritet) får tydligen det utseende som framgår av fig 3. 78b.

Den kompletta ASCII-koden framgår av fig 3. 78c. Bit 8 är alltid mark (dvs svart punkt) om paritet ej utnyttjas. Vid jämn paritet framgår bit 8 av tabellhuvudena (mark anges med vit text i svart ruta, space anges med svart text i vit ruta).

Tecken	Position 8	7	6	5	4	·	3	2	1
0	0	0	1	1	0	·	0	0	0
1	1	0	1	1	0	·	0	0	1
2	1	0	1	1	0	·	0	1	0
3	0	0	1	1	0	·	0	1	1
4	1	0	1	1	0	·	1	0	0
5	0	0	1	1	0	·	1	0	1
6	0	0	1	1	0	·	1	1	0
7	1	0	1	1	0	·	1	1	1
8	1	0	1	1	1	·	0	0	0
9	0	0	1	1	1	·	0	0	1
A	0	1	0	0	0	·	0	0	1
B	0	1	0	0	0	·	0	1	0
C	1	1	0	0	0	·	0	1	1
D	0	1	0	0	0	·	1	0	0
U	0	1	0	1	0	·	1	0	1
Y	0	1	0	1	1	·	0	0	1
Z	0	1	0	1	1	·	0	1	0

Jämn paritet (annars genomgående ettor i position 8)

a.



b.

Fig 3. 78 ASCII-koden

a. Exempel på binär utskrift av tecken i ASCII-kod (jämn paritet)

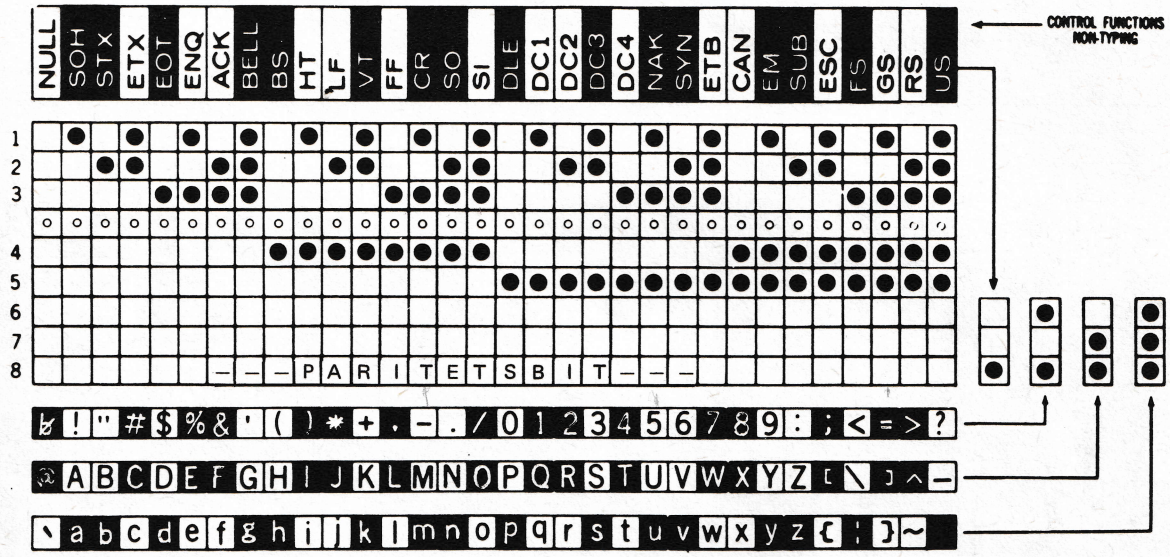
b. Ett tecken (U) i TTY:ns strömslinga.

ASC II TELETYPE CODES

Symbol	Hexadecimal Code	Symbol	Hexadecimal Code	Symbol	Hexadecimal Code
@	C0	W	D7	,	AC
A	C1	X	D8	—	AD
B	C2	Y	D9	.	AE
C	C3	Z	DA	/	AF
D	C4	[	DB	0	B0
E	C5	\	DC	1	B1
F	C6	]	DD	2	B2
G	C7	↑	DE	3	B3
H	C8	←	DF	4	B4
I	C9	NULL	00	5	B5
J	CA	BELL	87	6	B6
K	CB	↓	A0	7	B7
L	CC	!	A1	8	B8
M	CD	"	A2	9	B9
N	CE	#	A3	:	BA
O	CF	$	A4	;	BB
P	D0	%	A5	<	BC
Q	D1	&	A6	=	BD
R	D2	'	A7	>	BE
S	D3	(	A8	?	BF
T	D4	)	A9	CR	8D
U	D5	*	AA	LF	8A
V	D6	+	AB	RUBOUT	FF

c.

AMERICAN NATIONAL STANDARD CODE FOR INFORMATION INTERCHANGE (ASCII) X3.4-1968



d.

3.78c. TTY:ns tecken skrivna i hexadecimal kod (utan paritet, dvs 1 i pos 8)

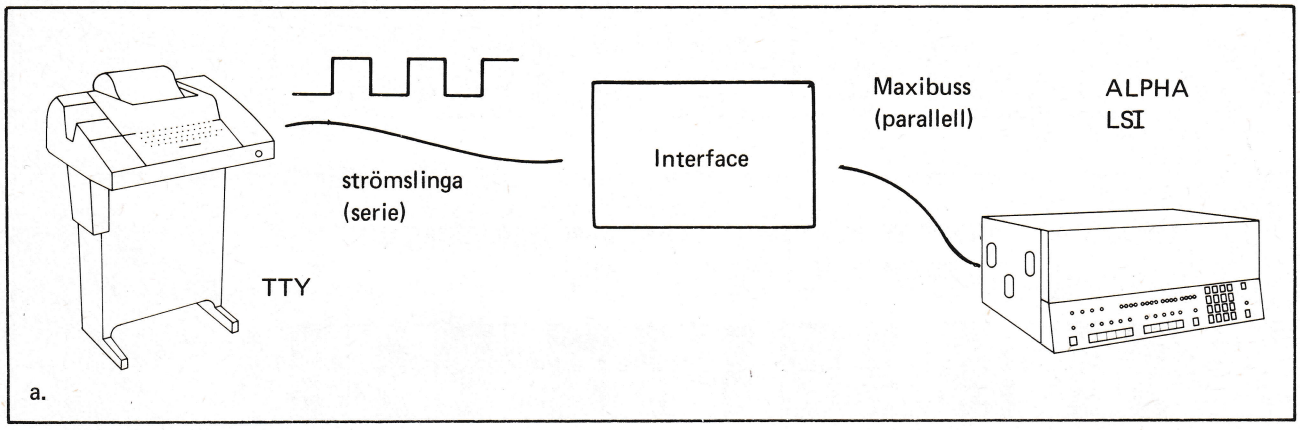
d. Kompletta ASCII-koden (tecken med mark som paritetsbit markerade med svart ruta).

3.3.4 TTY-INTERFACET

Vi har nu lärt känna TTY:n och dess funktion i stora drag. Från TTY:n kommer en strömslinga med signaler i serieform, kodade i ASCII-kod. Men vår minidator vill ha signaler i parallell form på databussen - och i binärkod. Hur skall ett sådant problem kunna lösas? Fig 3.79a.

Problemet är typiskt för dagens teknik. En stor del av teknikernas arbetsuppgifter består just i att bygga interface - dvs "kopplingsboxar" - mellan allehanda apparater och datorer. Det finns en speciell "Interface Manual" till ALPHA LSI som beskriver hur man kan gå tillväga och hur man ska klara av "handskakningen" (handshake) med ALPHA LSI:s Maxibuss.

När det gäller interface till TTY:n är problemet enkelt. TTY:n är ju ett standardtillbehör till minidatorer och därför finns som regel färdiga TTY-interface som "option",



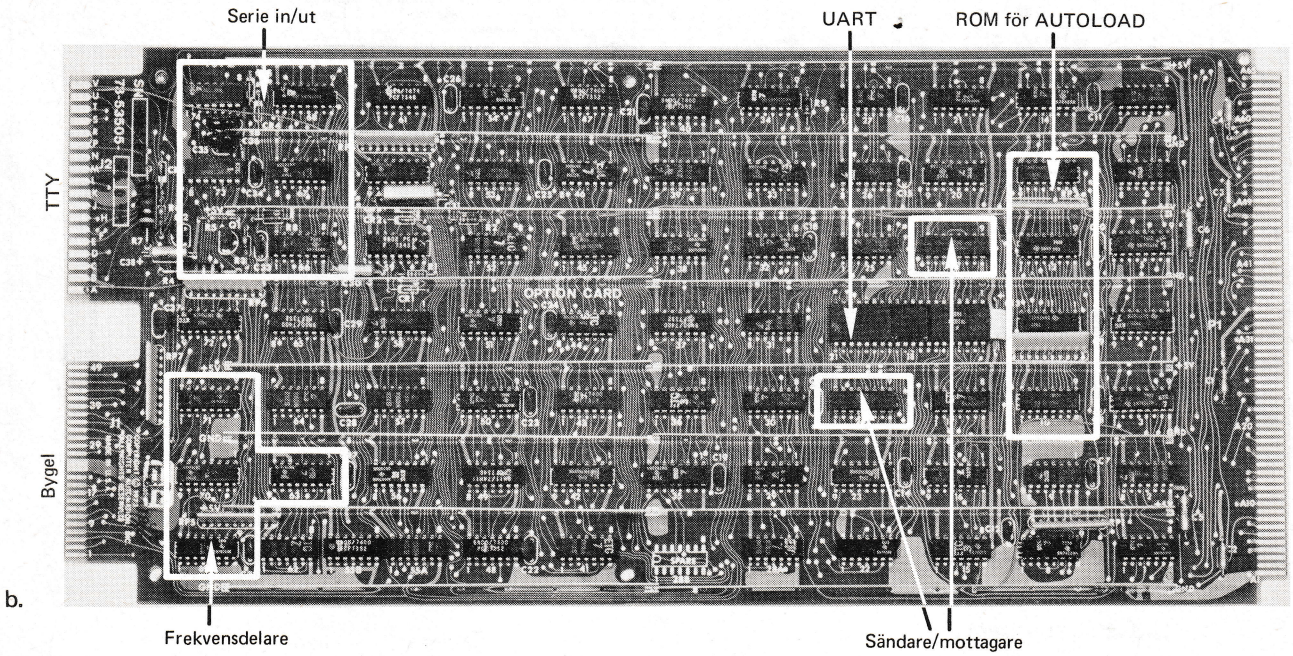


Fig 3.79 TTY-interfacet till ALPHA LSI.

dvs som tillbehör mot extra kostnad. Fig 3.79b. Många kringutrustningar arbetar med "TTY-snitt" (dvs med samma signalsystem och kod som TTY:n) av den anledningen att det finns TTY-interface tillgängliga på marknaden. Bildskärmar är ett exempel - men man ökar då ofta hastigheten exempelvis till 4800 baud.

3.3.4.1 Elektroniken

Låt oss studera TTY-interfacet till ALPHA LSI litet mera i detalj. Det innehåller en rad intressanta kretsar. Fig 3.79b.

Interfacet kommunicerar med TTY:n via fem ledningar. Två av dessa utgör strömslingan som normalt har strömmen 20 mA. Varje tecken överförs på denna slinga i form av ett antal strömavbrott på 9,09 ms. Varje tecken börjar som vi sett med ett avbrott (= startbit), fortsätter med 8 tidsmoment av ström eller avbrott (data) samt slutar med två tidsmoment ström (= stoppbitar).

Förutom strömslingan finns två kontrolledningar och jord.

Interfacet avkänner strömmen i slingan genom att känna spänningen över ett litet motstånd (R) som ligger inlänkat i slingan. Fig 3.80.

Interfacet kan signalera till TTY:n genom att bryta strömmen. Det är av denna anledning som en transistor (T) ligger inlänkad i slingan. Transistorn är normalt ledande (bottnad, dvs i till-läge). Vid signalering stryps transistorn under de tidsintervall som skall vara "spaces".

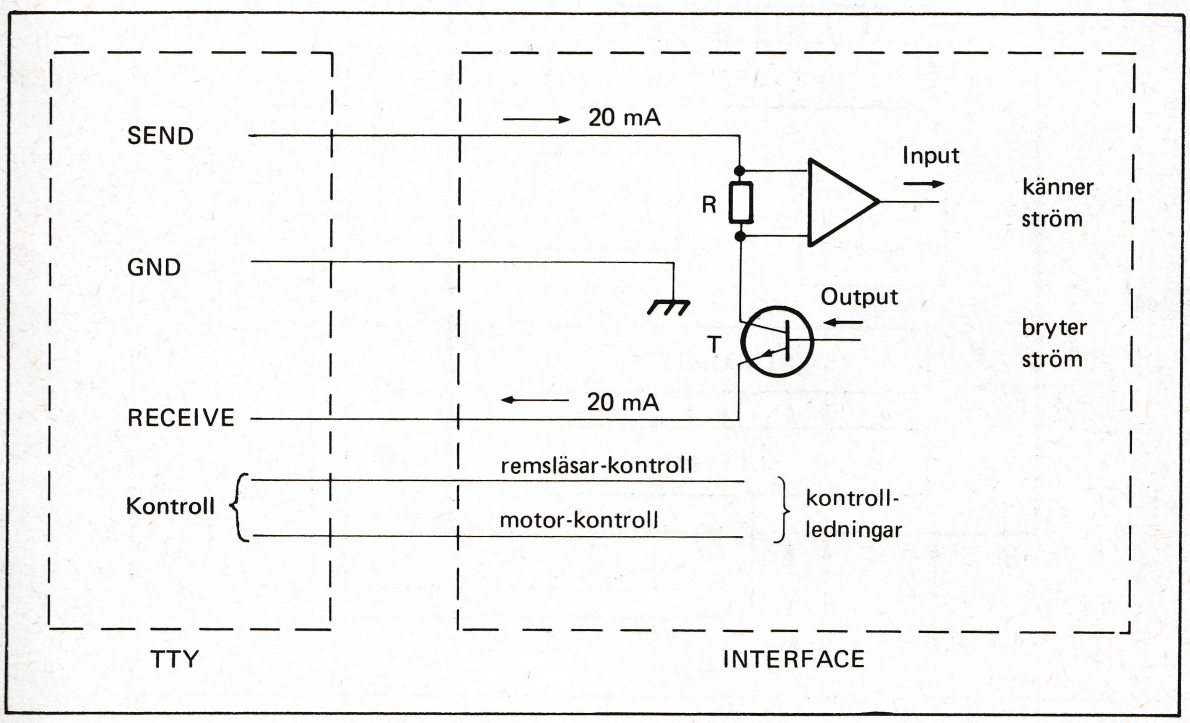


Fig 3.80 Ledningarna mellan TTY och interface.

Med spänningen över R och avbrott genom T kan tydligen slingan användas för signalering i båda riktningar.

Serieinformationen i strömslingan måste omvandlas till parallellinformation för maxibussen. Denna serie-parallell omvandling utförs i en LSI-krets med inte mindre än 40 ben. Vi ser den stora LSI-kretsen något till höger på fig 3.79b. Kretsen kallas UART, vilket betyder Universal Asynchronous Receiver Transmitter.

Innehållet i en UART (Texas Instruments TMS 6011) visas schematiskt i fig 3.81. Inkommande seriedata matas in i det övre skiftregistret och matas därefter över till ett buffertregister (en åtta bitars latch) i avvaktan på utläsning till databussen.

Parallellinformationen från databussen placeras i nedre buffertregistret och överförs vid startorder till nedre skiftregistret. Därifrån skiftas data ut i serieform till transistorn T som bryter strömslingan för varje nolla (dvs varje "space").

Det finns en rad kontrollingångar på vår UART, bl a en klockpulsingång som styr takten vid in- och utmatning av seriedata.

Vår UART har kretsar för generering av paritetsbit vid sändning och för paritetskontroll vid mottagning. UART:en avger automatiskt startbit och stoppbitar vid sändning och silar bort dessa bitar vid mottagning.

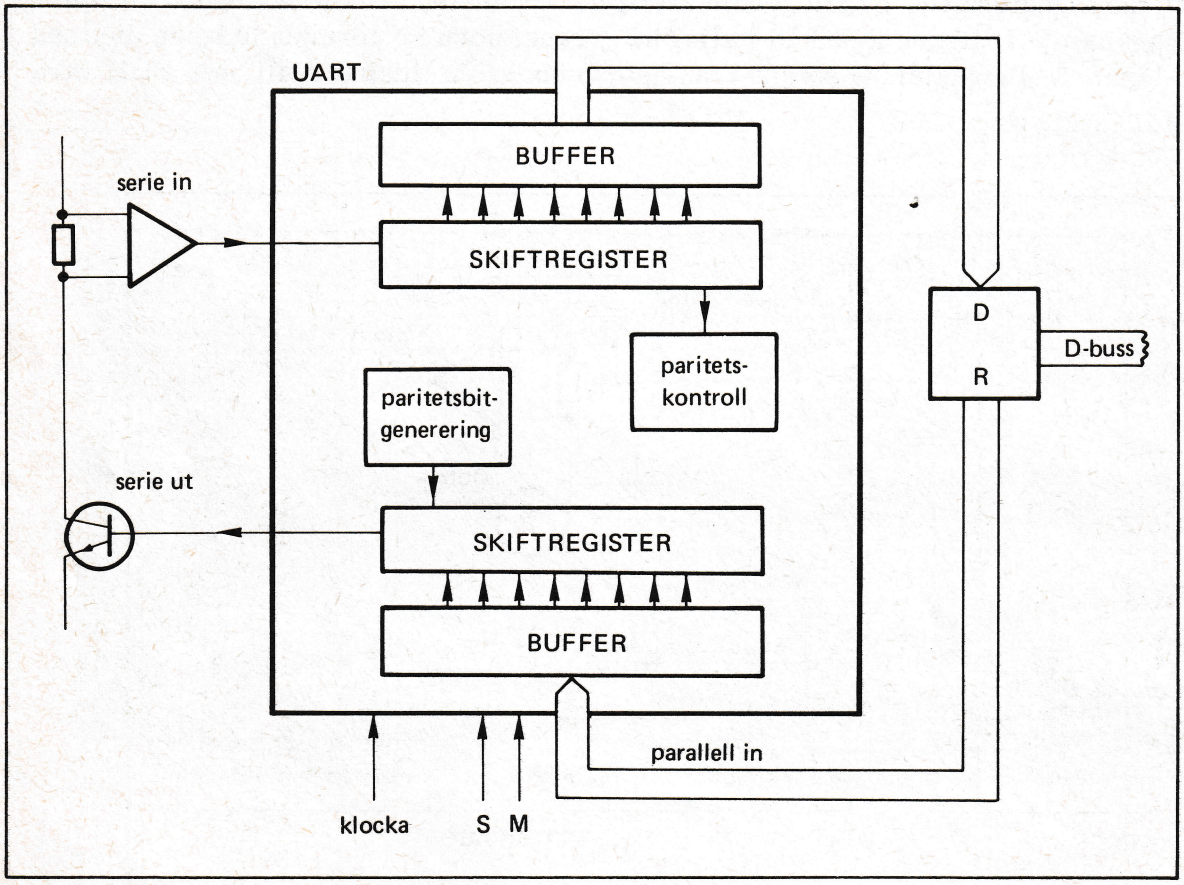


Fig 3.81 UART kallas serie-parallell omvandlaren i TTY-interfacet (S = sänd, M = mottag).

Vår UART måste klockas med 16 gånger högre frekvens än bithastigheten på serieingång och serieutgång. 16 gånger 110 bit/sek betyder en pulsfrekvens på ca 2 kHz. Det behövs dessutom pulser som avgränsar varje tecken (om 11 bitar).

En stor del av interfacekortet på fig 3.79b upptas av vippor och räknare. De räknar ner en klockpulssignal på 1 MHz som CPU:n utsänder via en ledning i kontrollbussen, och man får härigenom lämpliga klockpulsfrekvenser.

Fig 3.82 visar TTY-interfacets blockschema. För nerräkning av klockfrekvensen 1 MHz till ca 2 kHz (dvs ca 500 ggr) krävs en räknare om 9 bitar ($2^9 = 512$). På fig 3.79b ser vi tre kapslar som vardera innehåller 4 bitars räknare (74161) och som fungerar som frekvensdelare.

På kortet sitter totalt 24 vippor och 8 fyrabitars räknare.

Från övre buffertregistret i UART går en 8 bitars buss till en DRIVER/RECEIVER av samma typ som vi tidigare mött i CPU:n (fig 3.44). I läge DRIVE matas 8 bitars ord ut via denna buss till D-bussen. I läge RECEIVE kopplas data från D-bussen in till nedre buffertregistret i UART.

TTY-interfacet innehåller (liksom alla I/O-enheter) en adressavkodare. Den avkodar adressen på A-bussen. När TTY:n är adresserad kopplas såväl kontroll som SEND/RECEIVE-kretsarna in. Via C-bussen får interfacet order om sändning eller mottagning - eller ger CPU:n information om att data är tillgängliga.

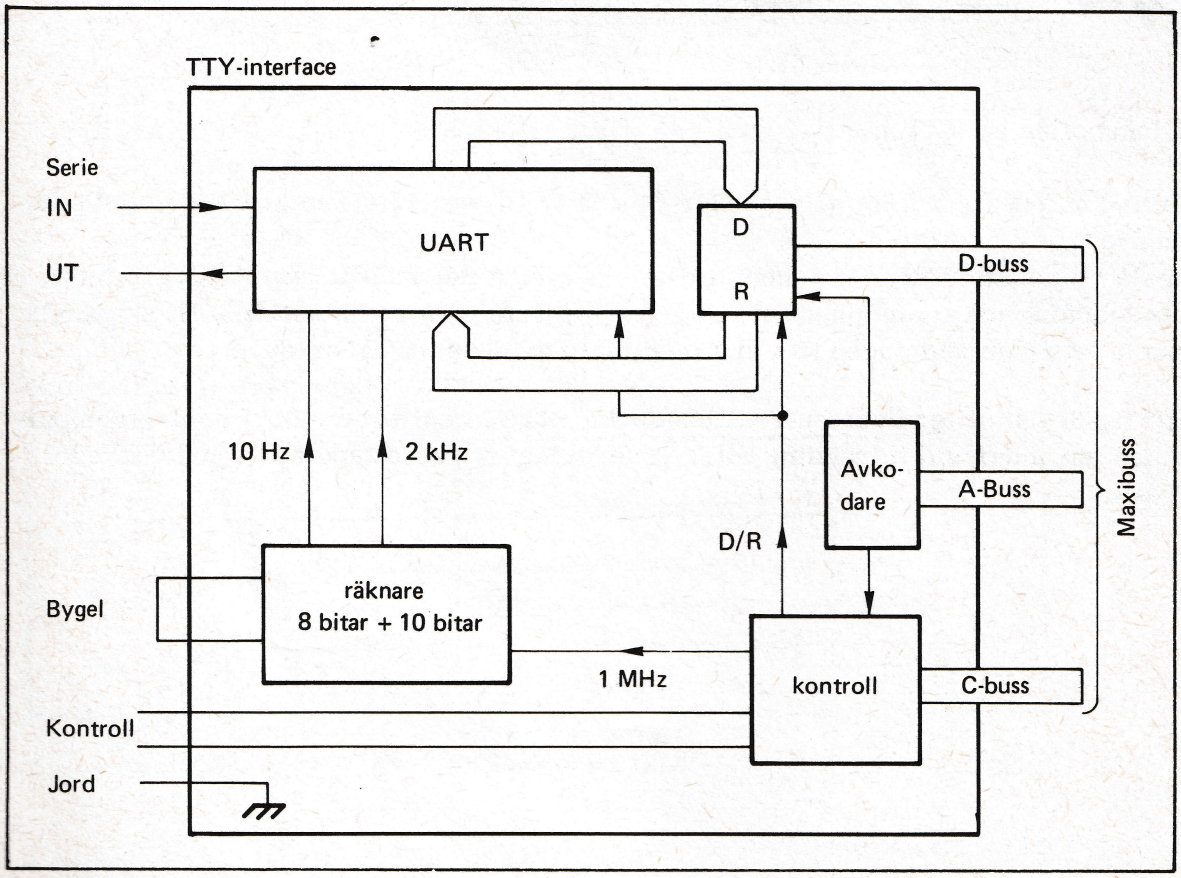


Fig 3.82 TTY-interfacets blockschema.

På TTY-kortet finns en kortkontakt med vars hjälp frekvensdelaren kan "byglas" för olika hastighet. Därmed kan interfacet användas även för bildskärmsterminaler som kör med högre hastighet. Fig 3.83.

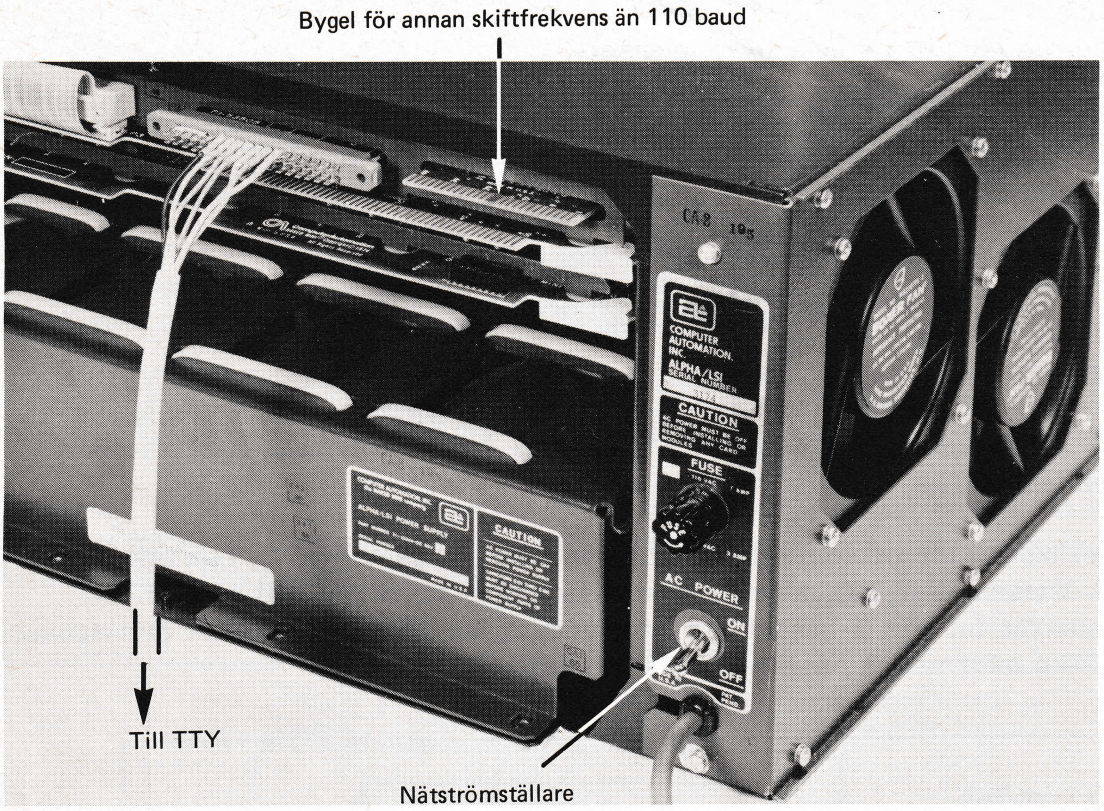


Fig 3.83 Inkoppling av TTY:n.

På interface-kortet, fig 3.79b, (Processor Option Card) ser vi ett antal IC-kapslar som är markerade AUTOLOAD.
Dessa kapslar innehåller ROM-minnen. I dessa har man permanent lagrat program för automatisk laddning (därav namnet autoload) av program från remsläsaren. Vi ska snart se hur enkelt programladdningen kan ske med hjälp av dessa ROM-minnen.

På kortet i fig 3.79b sitter även en realtidsklocka (RTC, real time clock) som avger pulser vilka kan användas vid tidräkning eller generering av avbrottspulser (med önskade tidsintervall).

3.3.4.2 Programstyrning av TTY:n

Nu är det dags att koppla in TTY:n till vår ALPHA LSI. Fig 3.83 visar inkopplingen. TTY:n inkopplas till det interfacekort vi beskrivit i fig 3.79b.

Innan vi nu kan köra igång måste vi en stund ägna oss åt programmeringsarbete.

3.3.4.2.1 Ett enkelt utskriftsprogram

ALPHA LSI har 33 I/O-instruktioner i sin instruktionslista. Det finns alltså en hel del att välja på. Dessa instruktioner är indelade i fyra grupper:

Sense (avkänning)
Select (val)
Input
Output

Datorer som skall kunna användas som komponenter i större system måste ha kraftfulla I/O-rutiner. Det har ALPHA LSI och används därför i stort antal på OEM-marknaden. (OEM = original equipment manufacturer).

Vi ska här enbart visa principen för användningen av I/O-rutiner, och vi kommer att använda följande I/O-instruktioner

SEL, 40XX, Select function
Med SEL adresserar man en I/O-enhet (exempelvis TTY) och ger order om den funktion som skall inkopplas (exempelvis tangentbord eller remsläsare).
WRA, 6DXX, Write from A-register.
WRX, 6FXX, Write from X-register.
Med dessa instruktioner avkänner man om I/O-enheten är redo och när I/O-enheten ger klarsignal (SER på kontrollbussen) överförs ett ord på 16 bitar från A- eller X-registret till I/O-enheten. Man kallar dessa instruktioner för "conditional output" eftersom CPU:n först kontrollerar om I/O-enheten är redo och tar detta som villkor för utskrift.

Formatet för en I/O-instruktion framgår av fig 3.84.

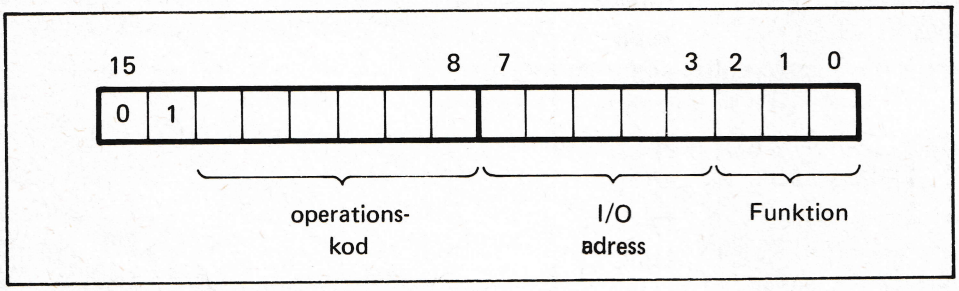


Fig 3.84 Format för I/O-instruktion.

Vissa I/O-enheter har fast adress. I ALPHA LSI har TTY:n adressen 7. På varje adress finns sedan 3 bitar (dvs 8 alternativ) för val av funktion.

Instruktionen SEL 7, 1 betyder exempelvis val av I/O-enhet 7 (TTY) och funktion 1 (tangentbord). Instruktion SEL 7,4 anger samma I/O-enhet (TTY:n) men funktion 4 (TTY-interfacet inkopplas och dess kontroll-vippor nollställs, "Initialize TTY Interface"). Maskinkoden får följande utseende:

$$\text{SEL } 7.1 = \underbrace{\text{Ø1ØØØØØØ}}_{\text{SEL}}\ \underbrace{\text{ØØ111}}_{7}\ \underbrace{\text{ØØ1}}_{1} = 4Ø39_{16}$$

Och på motsvarande sätt:

$$\text{SEL } 7,4 = 4Ø3C_{16}$$

Vi kallar i fortsättningen instruktionen SEL 7, 4 för "initiera TTY:n".

Man kan ge skrivkommando med instruktionen WRA 7, 1, där 7 liksom tidigare är TTY-adress och 1 anger funktion (i detta fall att avkänna om TTY-interfacet är redo att ta emot och om så är fallet överföra data från CPU:ns ackumulator (A) till buffertregistret på interfacekortet).

Låt oss nu praktiskt pröva en enkel rutin för utskrift. Vi har tidigare kört "demonstrationsprogrammet"

PC	MEMO	HEX
Ø	LDA 4	BØØ4
1	ADD 5	88Ø5
2	ST 6	98Ø6
3	HLT	Ø8ØØ
4	3 } data	ØØØ3
5	4	ØØØ4
6	Ø	ØØØØ

Låt oss nu göra en utskrift av resultatet innan vi stoppar datorn med HLT-instruktionen. Det ligger nära till hands att pröva med programmet i fig 3.85.

PC	MEMO	HEX
Ø	LDA4	BØØ4
1	ADD 5	88Ø5
2	ST 6	98Ø6
3	JMP 7	FØØ7
4	} Data	ØØ3E
5		ØØØ1
6		ØØØØ
7	SEL 7, 4	4Ø3C
8	WRA 7, 1	6D39
9	HLT	Ø8ØØ
eller		
9	JMP Ø	FØØØ

Fig 3.85 Ett program för utskrift på TTY:n

Istället för stoppinstruktionen HLT i programsteg 3 har vi nu en hoppinstruktion JMP 7, dvs hoppa till 7. I programsteg 7 initierar vi TTY-interfacet på det sätt som vi tidigare beskrivit och i programsteg 8 sker utskriften av innehållet i ackumulatorn (där resultatet från vår beräkning fortfarande är kvar).

Vi skall i detta fall addera talen $3E_{16}$ och 1_{16}. Resultatet bör alltså rimligtvis bli

$$\begin{array}{r} 3E \\ +\ 1 \\ \hline \text{Summa:}\quad 3F_{16} \end{array}$$

Stoppinstruktionen ligger i programsteg 9.

Knappa nu in programmet i ALPHA LSI på samma sätt som vi gjort tidigare (Glöm inte att kontrollera att programmet blivit rätt inskrivet!). Ställ sedan TTY:n i läge LINE (det ska surra från motorn).

Ställ programräknaren på Ø och starta datorn på känt sätt (dvs släck både STOP- och RUN-lamporna och starta med tryck på RUN-knappen).

Blev resultatet förvånande? Medan vi funderar på detta, kan vi modifiera programmet genom att byta ut stoppinstruktionen i programsteg 9 mot hoppinstruktionen JMP Ø. Vi startar datorn på nytt (med PC = Ø förstås!).

Nu har vi ingen stoppinstruktion med i programmet. Så fort datorn arbetat igenom programmet måste den hoppa tillbaks till början (JMP Ø) och börja om. Den räknar outtröttligt ut samma summa och skriver ut - om och om igen. Om allt fungerar som det ska så får vi nu höra "skramlet" från alla mekaniska detaljerna i TTY:n. När raden är fullskriven står TTY:n och stampar i högra papperskanten. Den har ju inte fått någon annan order och den skriver därför blint vidare mot marginalen. Vi kan stoppa datorn med ett tryck på kontrollpanelens STOP-knapp.

Vi kontrollerar resultatet på minnesadress 6. Det är som sig bör $3F_{16}$. Varför fick vi inte detta utskrivet på TTY:n?

Javisst ja! TTY:n förstår ju bara ASCII-kod och inte binärkod. Ser vi efter i fig 3.78 så hittar vi koden för frågetecknet. Fig 3.86 visar hålremsan och utskrift. (Om vi försöker med additionen BE + 1 = BF i vårt program så kan det hända att vi får samma utskrift. Hur är det då med pariteten? - Tydligen används inte paritetsbiten).

Nu har vi alltså fått förklaringen på våra frågetecken! Det här med koder är ett arbetssamt kapitel. Men vi ska snart se att det finns hjälpmedel!

För att få litet träning i både programmering och ASCII-kod ska vi nu göra ett både lärorikt och roligt program.

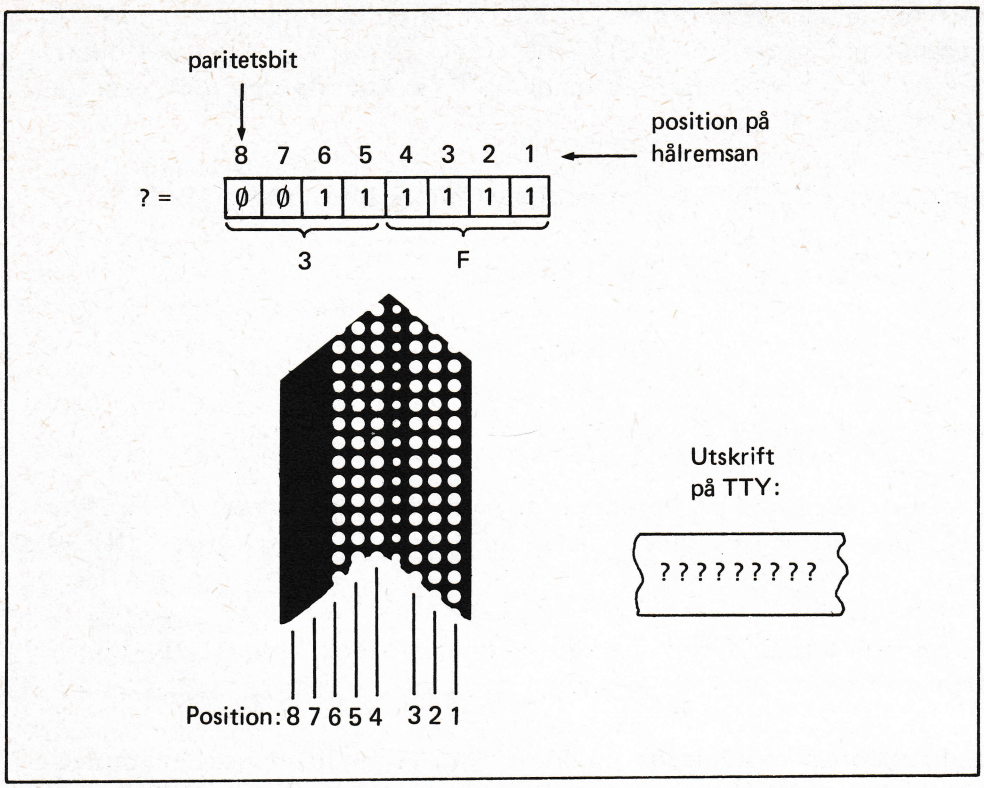


Fig 3.86 Talet $3F_{16}$ och dess tolkning av TTY:n

3.3.4.2.2 Ett program om ASCII-koden

Låt oss göra ett program som först nollställer ackumulatorn A och sedan upprepar följande rutin:

Skriv ut A
Inkrementera A
Skriv ut A
Inkrementera A
etc

A-registret kommer på detta sätt att genomlöpa alla binära värden från Ø till FFFF. Vid varje utskrift går de 8 minst signifikanta bitarna (dvs den minst signifikanta byten) till TTY-interfacet för utskrift. Vi får alltså i tur och ordning TTY:ns reaktion på binära tal i stigande följd.

Nu har ju TTY:n inte förstånd att själv göra en snygg uppställning av sin utskrift. Vi måste alltså ge TTY:n lämpliga direktiv för "ny rad" (radframmatning, line feed, LF) och för "vagnretur" (carriage return, CR). Eftersom typhuvudets indelning är binär (8 tecken till vänster, 8 till höger och fyra vertikala positioner) så låter vi här TTY:n skriva 8 tecken på varje rad. För att klara "formateringen" ska vi använda indexregistret (X) för att hålla reda på antalet tecken per rad.

Program kan skrivas på många sätt. Det är detta som gör programmeringen till en stor konst med obegränsade variationsmöjligheter. Det enkla problem vi ställt ovan, kan lö-

sas på många sätt och vi ser ett förslag i fig 3.87. Vi ska nu gå igenom programmet i detalj.

CP	MEMO	HEX	KOMMENTAR
Ø	LAP Ø	C6ØØ	Ø → A
1	LXP 8	C4Ø8	ladda X med 8
2	SEL 7,4	4Ø3C	kalla på TTY:n
3	WRA 7,1	6D39	skriv A → TTY
4	IAR	Ø15Ø	A + 1 → A
5	DXR	ØØA8	X - 1 → X
6	JXN $-3	3843	hopp om X ≠ Ø
7	LXP ØA	C4ØA	om X = Ø, ny rad → X
8	WRX 7,1	6F39	skriv X → TTY (ny rad)
9	LXP 8D	C48D	ladda X med 8D
10	WRX 7,1	6F39	skriv X → TTY (återgång)
11	JMP 1	FØØ1	hoppa till 1

Fig 3.87 Ett program om ASCII-koden

Vi har tidigare använt instruktionen LDA, dvs ladda A med innehållet i den cell som anges av instruktionens operandadress. LAP är, liksom LDA, en "ladda A" - instruktion. LAP är emellertid en direktinstruktion (immediate) som innehåller operanden (och alltså inte operandens adress) i instruktionen. LAP Ø betyder alltså: "ladda A med talet Ø". LDA Ø betyder däremot: "ladda A med talet i cellen med adressen Ø".

Vårt program i fig 3.85 börjar alltså med att ladda ackumulatorn A med Ø och indexregistret X med talet 8.

Vi initierar sedan TTY-interfacet med SEL 7,4 och skriver ut innehållet i A (dvs talet Ø) i TTY:n. Vi kan slå upp ASCII-koden i fig 3.78 och ta reda på vilket tecken som bör komma ut från TTY:n.

Talet Ø är enligt ASCII-koden en funktion som inte skrivs ut av teckenskrivaren. Nåväl - vi går vidare i programmet!

IAR är en instruktion som inkrementerar A, dvs adderar 1 till dess innehåll. DXR dekrementerar på motsvarande sätt innehållet i X. När IAR och DXR är exekverade blir alltså innehållet i A talet 1 medan innehållet i X blir 7.

JXN är en hoppinstruktion av det slag vi tidigare diskuterat i avsnitt 3.2.7. Hoppet sker här relativt programräknaren ($) och -3 anger "tre steg bakåt". Villkoret för hopp är att X ≠ Ø. Om X = Ø fortsätter datorn med efterföljande instruktion. I vårt fall är X = 7 och därför hoppar vi tillbaks till programsteg 3 (dvs 6 - 3) och skriver där ut A:s innehåll som nu är en etta på TTY:n. Ettan hittar vi i fig 3.78 och även ettan är en funktion som inte skrivs ut av teckenskrivaren.

När talet ett "skrivits ut" till TTY:n (även om det ej påverkat teckenskrivaren) inkrementeras ackumulatorn (A) och dekrementeras X-registret. Det blir ett nytt bakåthopp (JXN -3) och därefter utskrift av talet 2 till TTY:n.

När vi cirkulerat åtta gånger i slingan med bakåthopp, så har innehållet i indexregistret blivit noll och därmed gäller inte längre hoppvillkoret (JXN). Vi går alltså vidare till programsteg 7 som laddar X direkt med talet ØA. Varför?

I fig 3.78 hittar vi ASCII-koderna för kontrollfunktionerna LF (line feed) och CR (carriage return):

LF: ØØØØ1Ø1Ø = $ØA_{16}$

CR: 1ØØØ11Ø1 = $8D_{16}$

Laddar vi X-registret med ØA och sedan skriver ut tecknet får vi alltså ny rad. Laddar vi sedan X-registret med 8D och skriver ut på nytt, får vi vagnretur. Där har vi alltså orsaken till instruktionerna

```
LXP ØA
WRX 7,1
LXP 8D
WRX 7,1
```

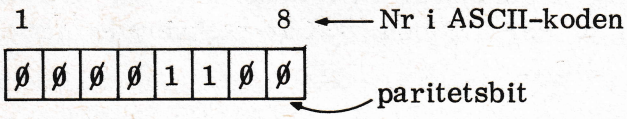
Och nu hoppar vi alltså tillbaks till programsteg 1 i vårt program. (Observera, inte till Ø!). När vi senaste vändan passerade programsteg 4 ökades ackumulatorns innehåll till 8. Vi startar alltså nu på ny rad längst till vänster med utskrift av ASCII-kodens tecken för talet 8. Även detta är en funktion som inte får teckenskrivaren att reagera.

Och så fortsätter utskriften. Vi får komma ihåg, att TTY:n inte reagerar med utskrift för kontrollfunktioner i ASCII-koden. Därför får vi till att börja med några lustiga hopp i utskriften.

När vi räknat upp innehållet i A till FF så blir nästa värde i de 8 minst signifikanta bitarna ØØØØØØØØ. Vi börjar därför om från början igen och förloppet upprepas.

Utskriften fortsätter så länge papperet räcker (om vi inte trycker på STOP på kontrollpanelen) och en del av utskriften visas på fig 3.88.

När reagerar då teckenskrivaren? Låt oss se efter vilket tal som motsvarar siffran 0! Det är enligt fig 3.78.



I datorns register ligger ASCII-kodens första bit som LSB (minst signifikant siffra) och med hänsyn till positionsvärdena får vi därmed värdet

$0 \cdot 1 + 0 \cdot 2 + 0 \cdot 4 + 0 \cdot 8 + 1 \cdot 16 + 1 \cdot 32 + 0 \cdot 64 + 0 \cdot 128$

dvs

$16 + 32 = 48$

I fig 3.88 har vi markerat talvärden för de första tecknen i varje rad. Vi ser att TTY:n är så utförd att den trycker versaler för alla inkommande bokstavskoder (såväl versaler som gemener).

```
        HIJKLMNO
        PQRSTUVW
120     XYZ[\]↑
0
8
16
24
32       !"#$%&'
40      ()*+,-./
48      Ø1234567
56      89:;<=>?
64      @ABCDEFG
72      HIJKLMNO
80      PQRSTUVW
88      XYZ[\]↑←
96      @ABCDEFG
104     HIJKLMNO
112     PQRSTUVW
120     XYZ[\]↑
```

Fig 3.88 TTY:ns utskrift av binärkod för talen 0 till 127.

3.3.4.2.3 En enkel läsrutin

Vi har tidigare sett hur datorn kan skriva ut data på TTY:n med hjälp av instruktionen WRA, dvs "skriv från ackumulatorn A". För att få utskriften att fungera, måste vi först initiera TTY-interfacet med en SEL-instruktion (select). WRA-instruktionen sköter sedan om signalerna på kontrollbussen till TTY-interfacet och detta i sin tur sköter om kontrollsignalerna från interfacet till själva TTY:n.

Först när TTY:n signalerar "klart för utskrift" så sänds data över via interfacet till TTY:n. WRA-instruktionen är alltså en "conditional word output instruction" som släpper iväg data för utskrift först sedan klarsignal erhållits från I/O-enheten.

Motsvarande läsinstruktion heter på MEMO-språket RDA (dvs läs data till ackumulatorn A). Den fungerar likartat med WRA och är alltså en "conditional word input instruction". När det gäller läsning av data tillkommer ett problem. Vi har tidigare kört TTY:n i läge LOCAL och därmed automatiskt fått utskrift (på teckenläsare eller hålstans) av de tecken vi slagit in på tangentbordet.

När vi sätter TTY:n i läge LINE kopplas signalerna från den roterande omkopplaren (parallell-serieomvandlaren i TTY:n) till TTY-interfacet istället för till teckenskrivaren. Vi får alltså ingen utskrift av det vi skriver in på tangentbordet - vi tappar lätt kontrollen på vad vi skriver in.

Naturligtvis finns det rutiner som ordnar en kontrollutskrift av vad vi skriver in på tangentbordet. Det tillgår så att interfacet sänder tillbaks data till teckenläsaren. Detta kallas AUTO ECHO. Man initierar ekot med instruktionen SEL 7,Ø. Siffran 7 är som tidigare adressen till TTY-interfacet och Ø anger ENABLE AUTO ECHO, dvs ställer den kontrollvippa i interfacet vilken styr ekosignalen tillbaks till TTY:n.

Nu är vi mogna att skriva en enkel läsrutin med automatiskt eko. Den kan se ut på följande sätt:

PC	MEMO	HEX	Kommentar
Ø	SEL 7,4	4Ø3C	kalla på TTY-interfacet
1	SEL 7,Ø	4Ø38	ställ "eko-vippan"
2	RDA 7,1	5939	läs från tangentbordet
3	HLT	Ø8ØØ	stopp

Instruktionerna är uppbyggda på samma sätt som vi tidigare beskrivit. Vill man studera detaljerna är "Programming Referance Manual" en lämplig källa.

Ladda programmet i ALPHA LSI. Sätt TTY:n i läge "LINE". Nollställ P (dvs programräknaren PC) och starta på RUN.

Tryck nu på en tangent och kontrollera sedan innehållet i ackumulatorregistret (A). Vi ser nu ASCII-koden för det på tangentbordet nyss inskrivna tecknet.

För att bekvämare kunna läsa ASCII-koden kan vi göra ett program som automatiskt skriver ut ackumulatorns innehåll i kontrollpanelens dataregister. Det kan exempelvis se ut som på fig 3.89.

PC	MEMO	HEX	Kommentar
Ø	SEL 7,4	4Ø3C	kalla på TTY-interfacet
1	SEL 7,Ø	4Ø38	ställ EKO-vippan
2	SEN 7,1	4939	avkänn tangentbordet
3	JMP $-1	F6Ø1	om nej: hoppa 1 steg bakåt
4	RDA 7,1	5939	om ja: läs TTY till A
5	OCA	44Ø4	sätt A i panelens dataregister
6	JMP $-4	F6Ø4	hoppa 4 steg bakåt

Fig 3.89 Program för indikering av ASCII-koden på kontrollpanelen

I programmet ser vi tre nya instruktioner SEN, JMP $ och OCA. Vi ska börja med en förklaring.

SEN, (Sense and Skip on Response, 49XX) känner av en kontrollsignal från en I/O-enhet. Om kontrollsignalen är 1 överhoppas nästa programsteg. Om kontrollsignalen är Ø fortsätter programmet med nästa steg.

JMP $ (dvs "jump dollar") är vår tidigare hoppinstruktion JMP (unconditional jump) där vi med $ markerar att vi använder relativ adressering. JMP $-1 betyder bakåthopp ett steg i relation till programräknaren ($).

Om kontrollsignalen visar 1 (dvs att data finns tillgängliga) hoppar datorn över programsteg 3 och exekverar programsteg 4.

Om kontrollsignalen visar Ø (inga data tillgängliga) kommer efterföljande instruktion JMP$-1 att medföra ett bakåthopp. Datorn arbetar därmed i en slinga och roterar i

denna slinga ända tills kontrollsignalen från I/O-enheten signalerar att nya data är tillgängliga. Med SEN och JMP kan man alltså få datorn att vänta och vara beredd att ta hand om data, så fort kontrollsignal kommer.

Hoppinstruktionen i programsteg 3 kan vara JMP 2 eller JMP $-1 och båda ger samma resultat. Finessen med den relativa adresseringen i JMP $-1 är att programmet kan flyttas (relokeras) utan att funktionen ändras.

Man behöver inte heller hålla reda på absoluta adressen som ju ofta måste ändras när olika delar av ett program ändras. Vid absolut adressering (som i JMP 2) måste hoppadresserna ändras om programmet relokeras.

I fig 3.61 ser vi hur JMP $ instruktionen är uppbyggd. Grundkoden är FØØØ. Med relativ adressering bakåt (MM = 11) blir koden F6ØØ.

Med ett steg bakåt får vi slutligen

JMP $-1 F6Ø1

OCA (Output A to Console Data Register) är en instruktion som används när A-registret skall indikeras på kontrollpanelen. OCA har koden

$44Ø4_{16}$

När vi nu känner instruktionernas funktion är det lätt att se hur programmet i fig 3.89 arbetar.

Efter initiering av TTY:n (SEL 7,4) och inkopplingen av ekot (SEL 7,Ø) följer "vänta på data"-rutinen (SEN 7,1 och JMP $-1).

När en tangent på TTY:n anslås matas motsvarande tecken i serieform till TTY-interfacet och när det kommit på plats i buffertregistret går en kontrollsignal till CPU:n. Kontrollsignalen avkännes av SEN-rutinen som därvid låter programräknaren hoppa fram ett steg extra. JUMP-instruktionen överhoppas alltså och "väntan" är slut. Tecknet läses in till ackumulatorn (RDA 7,1) och kopieras till kontrollpanelens dataregister (OCA).

Programräknaren ställs nu tillbaka (JMP $-4) och CPU:n väntar därefter på nästa tecken från TTY:n.

Mata nu in programmet i ALPHA LSI och kör det! Kontrollera ASCII-koden i fig 3.78.

Vi har nu bekantat oss med maskinspråket hos ALPHA LSI. Vi har genom mikro- och maskininstruktionerna i detalj kunnat följa händelseförloppen (i varje fall "ana" dem) i de svarta plastkapslarna på kretskorten. Händelserna i TTY:n är ju mera akustiskt och mekaniskt påtagliga och där krävs inte samma abstraktion i tanken.

Det är både spännande och intressant att arbeta på detta maskinnära sätt (Det händer ju en massa saker som man inte väntat sig!). Skall vi försörja oss på programmering och användning av datorer, måste vi emellertid använda effektivare metoder än maskinprogrammering.

3.4 Software support

Vi har nu blivit rätt väl bekanta med hur vår dator fungerar rent tekniskt. Vi kan med hjälp av instruktionslistan skriva program som datorn utför, steg för steg enligt våra önskningar. Men vi har också upplevt att varje litet program kräver mycket arbete, detta gäller både för uppställningen av programmet (så att det blir korrekt) och för laddningen av programmet i datorn.

Mycket av detta programarbete kan vi låta datorn själv utföra med hjälp av olika programrutiner. Vi ska nu pröva några hjälpmedel av denna typ. Fig 3.90 visar en del av den programvara som finns tillgänglig för ALPHA LSI.

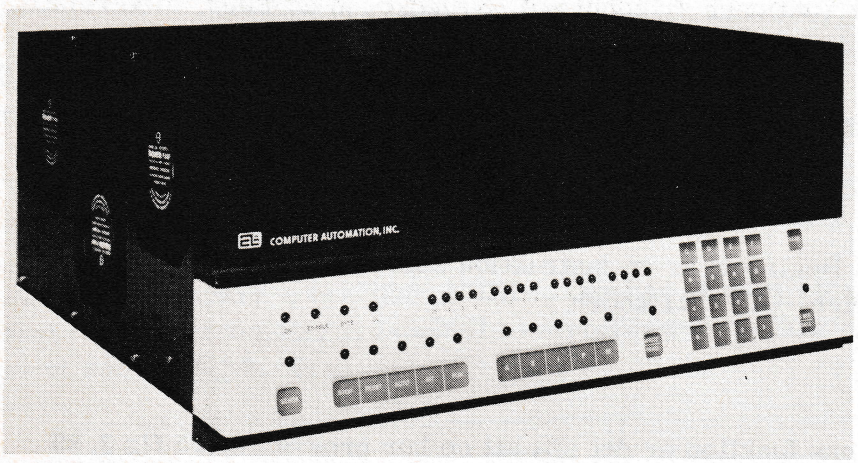


Fig 3.90 Maskinvara och programvara (hardware and software).

3.4.1 ASSEMBLERPROGRAMMERING

Ordet "assembler" låter främmande första gången man stöter på det i datorsammanhang. Ordet har grundbetydelsen "församla" (FN:s generalförsamling heter UN General Assembly), "sätta samman", "montera" m m.

När det gäller datorer är en assembler ett hjälpmedel för sammansättning (och "montering") av maskininstruktioner till ett fungerande program. En assembler är ett program (och alltså inte en apparat).

Till ALPHA LSI:s programvara hör en "konversationsassembler" som kallas OMEGA. Den är konverserande på så sätt att den vid olika tillfällen kommenterar vårt programarbete och talar om när vi begår "syntaktiska" fel, dvs uttrycker oss på ett sätt som datorn ej kan tolka.

Konversationsassemblern OMEGA finns i ett flertal olika varianter. Vi kommer att använda OMEGA2 som är avsedd för ALPHA LSI 2. Till ALPHA LSI:s "software support" hör även tre program med beteckningarna DEBUG, LAMBDA och BDP/VER.

DEBUG betyder avlusning. Det är en vanlig term inom elektroniken för felsökning och korrigering av fel. Begreppet DEBUG används även i samband med korrigering av programvara. Programmet DEBUG kan användas för korrigering och provkörning av programmen medan de ligger i minnet.

LAMBDA är namnet på en grekisk bokstav (λ) men används även som namn på ett program för laddning av remsor m m till ALPHA LSI. LAMBDA fungerar i stort som AUTOLOAD, dvs den laddningsrutin som finns inbyggd i TTY-interfacets ROM-kretsar.

BDP/VER, är en förkortning av binary dump/verify. Med hjälp av BDP/VER kan vi stansa ut remsor av program eller kontrollera att remsor stämmer överens med program som är lagrade i minnet.

OMEGA, DEBUG, LAMBDA och BDP/VER är fyra användbara program. De levereras i form av hålremsor, fig 3.91. Vi ska nu ladda dem i datorns minne med hjälp av AUTOLOAD.

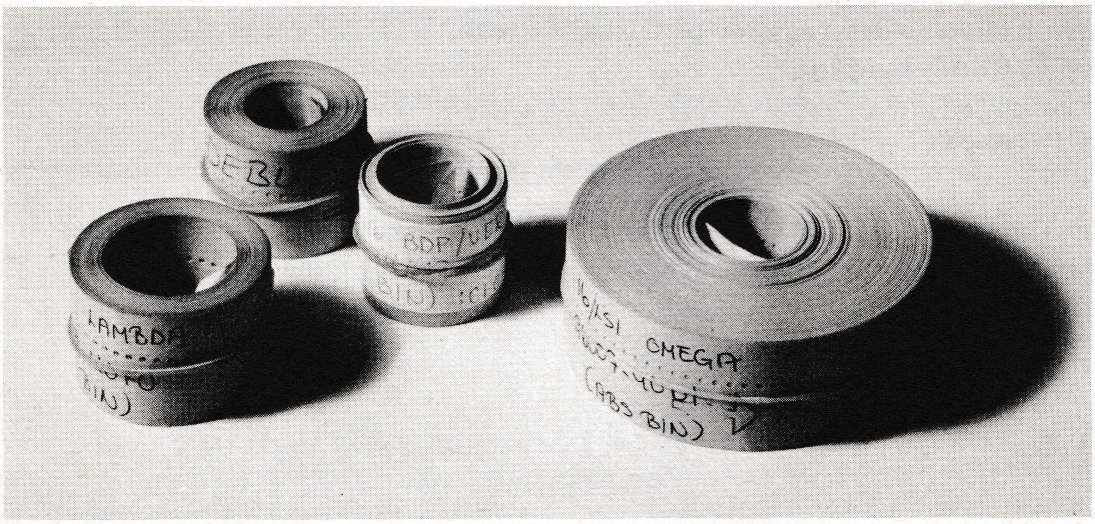


Fig 3.91 Fyra användbara hjälpmedel för assemblerprogrammering.

3.4.1.1 AUTOLOAD

AUTOLOAD är ett program som hämtar data från remsläsaren på TTY:n och placerar dem på önskad plats i datorns minne.

Om programmet skall lagras på den plats som anges på remsan (dvs laddas med absolut adress) sker autoladdningen enligt följande:

1. Slå till strömställaren för nätspänningen (power on) på ALPHA LSI:s baksida.
2. Ställ strömställaren på TTY:n i läge LINE.
3. Tryck på RESET.
4. Tänd STOP-lampan med STOP-knappen.
5. Tänd S REG/DATA-lampan, dvs koppla in SENSE-registret (S REG).
6. Ställ S REG i läge :Ø (dvs ØØØØ).
7. Släck STOP-lampan med STOP-knappen.
8. Ladda hålremsan i TTY:ns remsläsare, och sätt omkopplaren på remsläsaren i läge START.
9. Kontrollera att SENSE-lampan (längst till vänster på kontrollpanelen) är släckt.
10. Tryck på AUTO och remsan läses in.

Punkt 1-2 kopplar in nätspänningen.
Punkt 3-4 har vi mött tidigare i rutinen READY CONSOLE.
Punkt 5-6 nollställer SENSE-registret och meddelar därigenom CPU:n att programmet skall laddas med absolutadress från TTY:ns remsläsare.
Punkt 7-8 ställer CPU:n i läge RUN ENABLE och remsläsaren färdig för start.
Punkt 9 Om SENSE-lampan är tänd när AUTOLOAD exekveras startas programmet automatiskt efter inläsning. Önskar man enbart inläsning skall SENSE-lampan vara släckt.
Punkt 10 startar AUTOLOAD-programmet.

Vill vi ha programmet lagrat på annan plats - dvs relokerat - måste SENSE-registret ställas i läge :8 (dvs 1ØØØ) och önskad lagringsadress sättas i X-registret. Punkt 6 i ovanstående laddningsrutin måste då ändras till

6.1 Ställ S REG i läge :8.
6.2 Släck S REG/DATA-lampan.
6.3 Ställ kontrollpanelens dataregister på önskad adress.
6.4 Tänd WRITE/READ-lampan.
6.5 Tryck på X.
7 Släck STOP-lampan .... etc.

Vi har nu bekantat oss med AUTOLOAD-rutinen. Men vi kanske ska tänka oss för litet grann innan vi laddar in våra fyra hjälp-program. Får de plats i minnet? Var ska vi lägga dem?

3.4.1.2 Minnets disposition

Vår "konfiguration" av ALPHA LSI är en av de minsta varianterna med avseende på minneskapacitet. Den har 8k kärnminne. Vi måste se upp när vi nu gör upp en plan för programmens placering i minnet.

Programmens utrymmesbehov finns angivet på remsorna. Början på hålremsan med OMEGA-programmet visas i fig 3. 92.

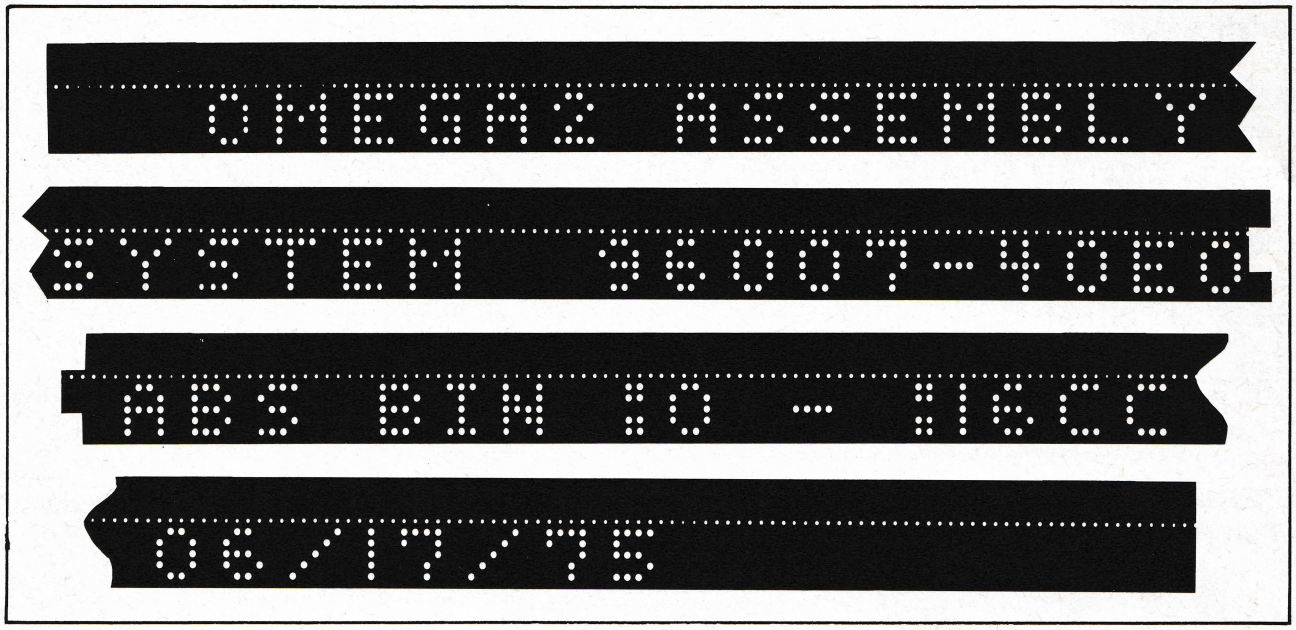


Fig 3. 92 Startremsan för OMEGA-programmet.

Som framgår av remsan ska programmet läggas i cellerna :Ø till :16CC (ABS BIN :Ø - :16CC). Det omfattar alltså totalt 5837_{10} celler och täcker även scratchpad.

Enbart OMEGA tar alltså drygt halva minnet i anspråk.

När OMEGA körs utnyttjar programmet både scratchpad (dvs cellerna :Ø till :FF) och en stor del av området ovanför :16CC. OMEGA måste placeras på den avgivna platsen för att kunna fungera korrekt. (ABS BIN = absolut binärt).

Vi har större frihet att placera de övriga programmen. Enligt remsorna kräver dessa följande utrymme.

LAMBDA	:23Ø	dvs	$56Ø_{10}$
BDP/VER	:BA	dvs	186_{10}
DEBUG	:2ØØ	dvs	512_{10}

Vi kommer att lägga in programmen på det sätt som visas i fig 3. 93. Man frågar sig kanske varför just på det här sättet. Och vad menas med SCRATCHPADKOPIA i cellerna :1DØØ till :1DFF?

Valet av minnespositioner är enkelt. OMEGA måste placeras på angiven plats och de tre övriga programmen har vi lagt in godtyckligt, enligt fig 3.93. På detta sätt har vi sparat ett område som vi fritt kan disponera dvs som är avsett för USER PROGRAM. Det omfattar cellerna :17ØØ till :18FF.

Från :1DØØ till :1DFF ligger 256 celler som kallas SCRATCHPADKOPIA. Varför?

SCRATCHPAD används både av OMEGA och LAMBDA för lagring av adresser m m. LAMBDA använder SCRATCHPAD för tillfällig lagring och dessa data behöver därför aldrig sparas.

OMEGA:s SCRATCHPAD innehåller däremot adresser m m som behövs varje gång OMEGA används. När vi första gången laddar in OMEGA i minnet lägger vi därför in en kopia av OMEGA:s SCRATCHPAD på ett ställe där det inte förstörs. Vi kallar därför detta område för SCRATCHPADKOPIA och det innehåller alltså en kopia av den del av OMEGA som skall ligga i SCRATCHPAD när OMEGA används. Varje gång vi skall använda OMEGA måste vi därför börja med att kopiera ner innehållet i SCRATCHPADKOPIA till SCRATCHPAD.

I fig 3.93 finns en minnesarea som kallas LAMBDA BUFFER. Det används av LAMBDA och för att undvika kollision har vi begränsat USER AREA till minnesarean :17ØØ till :18FF.

OMEGA använder även en del av minnesutrymmet nedanför :17ØØ i fig 3.93 men det ska vi återkomma till strax.

3.4.1.3 Laddning

Nu kan vi använda AUTOLOAD för inladdning av de fyra hålremsorna (LAMBDA, BDP/VER och DEBUG kan ev ligga på samma remsa, men har då stopptecken och mellanrum mellan programmen).

Vi börjar med OMEGA. Den skall laddas på den adress som anges av remsan och vi använder rutinen 1-10 i avsnitt 3.4.1.1. Remsan är lång och tar bortåt 25 min att ladda via remsläsaren. Man måste se upp så att remsan löper lätt, för annars kan remsläsaren på TTY:n läsa fel och stanna. Och då får man börja om från början! (Använd stora papperskorgar som remssamlare och var rädd om remsan så att den inte viks eller skadas).

När vi är färdiga med OMEGA-laddningen kommer turen till de övriga tre programmen. Vi börjar här med LAMBDA. På LAMBDA-remsan står adresserna :ACØ - :CEF och det stämmer inte med vår minnesdisposition. Vi måste alltså relokera och köra den alternativa relokeringsrutinen för punkt 6 i avsnitt 3.4.1.1.

När vi kommer till punkt 6 i rutinen ställer vi alltså SENSE-registret i läge :8 och sätter in den önskade startadressen för LAMBDA (dvs :1AØØ) i X-registret. Vi fullföljer sedan punkterna 7-10 i rutinen och har därmed laddat LAMBDA.

Laddningen av de övriga två programmen tillgår på likartat sätt. Startadresserna är här

BDP/VER	:1C4Ø
DEBUG (DBG)	:1EØØ

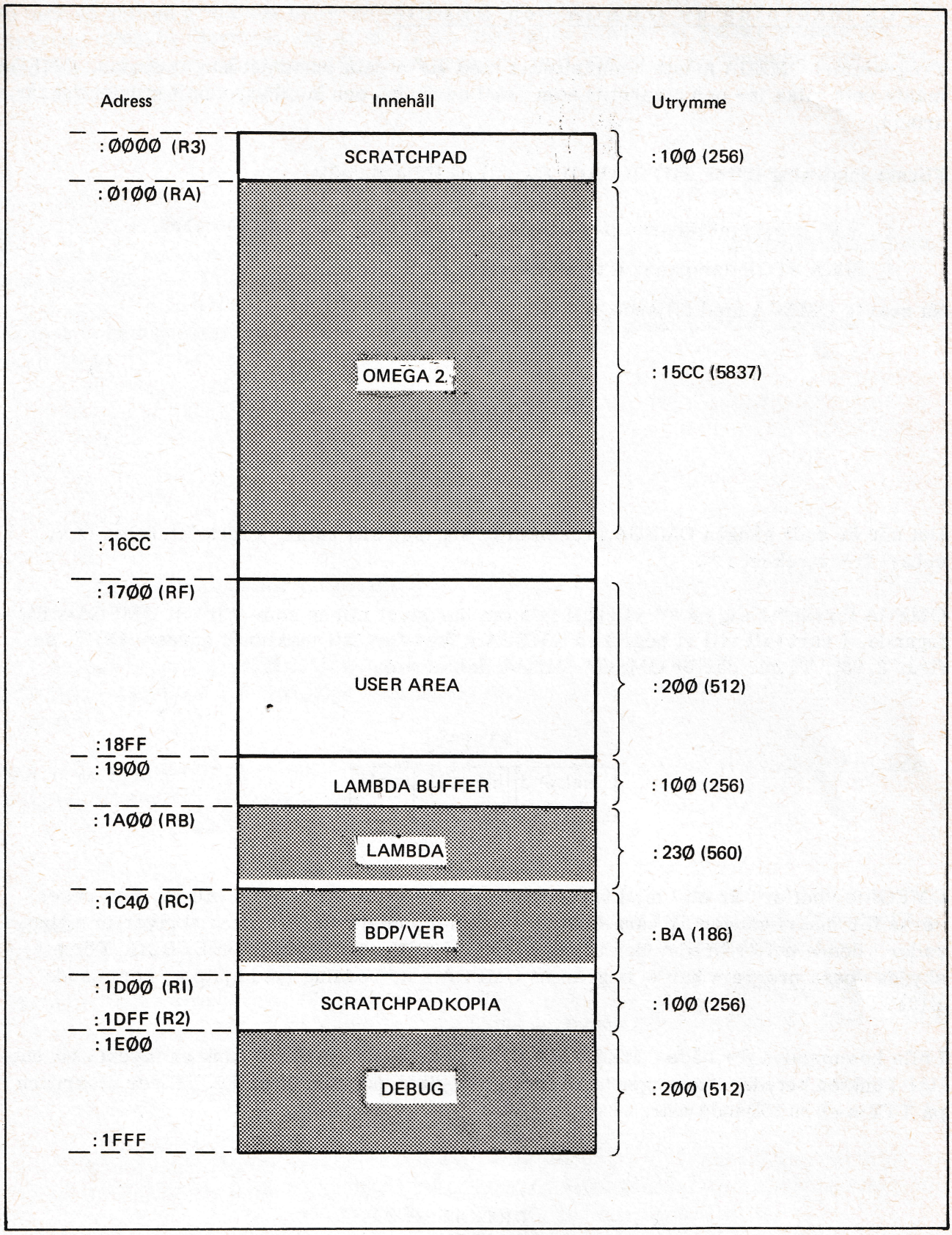


Fig 3.93 Minnesdisposition.

3.4.1.4 Inkörning av OMEGA och DEBUG

Programmet OMEGA måste kompletteras med vissa data innan det kan användas. OMEGA måste veta vilka in- och utenheter som skall användas och hur långt upp i minnet det får arbeta.

Första inkörning (efter AUTOLOAD) går till på följande sätt:

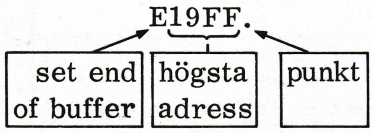
1. Ställ programräknaren på OMEGA:s startadress, dvs sätt P = :1ØØ.
2. Släck STOP-lampan och tryck därefter på RUN.

Nu svarar OMEGA med följande utskrift på TTY:n

```
OMEGA2 (EØ)
?
```

Det här är enda gången OMEGA presenterar sig med sitt namn. I fortsättningen få vi enbart frågetecknet (?).

OMEGA förväntar sig nu att vi skall tala om hur stort minne som står till OMEGA:s förfogande. I vårt fall vill vi begränsa OMEGA:s framfart till maximalt adress :19FF (se figur 3.93). Vi ger därför OMEGA följande kommando



OMEGA:s "buffer" är ett "mjukvaruregister", dvs ett antal celler i minnet som reserveras för inskrivningen av våra MEMO-symboler och direktiv. Om vi skriver in många rader i denna buffer breder den ut sig över LAMBDA, BDP/VER och DEBUG. För att skydda dessa program har vi begränsat OMEGA:s utbredning med hjälp av kommandot E19FF.

I alla kommandon för både OMEGA och DEBUG är den avslutande punkten mycket väsentlig! Punkten betyder "kom igen". Så fort vi slagit in punkten efter E19FF ser utskriften på TTY:n ut på följande sätt:

```
OMEGA2 (EØ)
?E19FF.
?
```

Vi har nu satt en gräns för bufferns utbredning i minnet. Men OMEGA vill ha ytterligare direktiv! OMEGA förväntar sig nu att vi ska ange vilka kanaler vi önskar använda för input (I), output (O) och listning (L). Det anger vi med "connect"-kommandon som i vårt fall kan ha följande utseende:

```
CI5.
CO1.
CL1.
```

I5 anger att input tas från minnet. O1 anger att output kommer via remsstans och L1 att listningen görs på TTY. (Se vidare Appendix 1).

Så fort vi satt punkt efter ett kommando svarar OMEGA med ett nytt frågetecken. Utskriften på TTY har nu följande utseende

```
OMEGA2 (EØ)
?E19FF.
?CI5.
?CO1.
?CL1.
?
```

Nu är OMEGA inkörd (eller initierad) och det är dags att komplettera DEBUG med några praktiska "register".

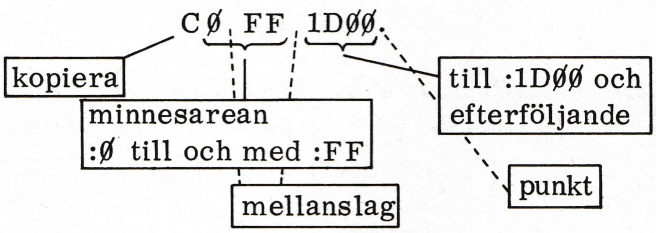
Hur hoppar vi nu över från OMEGA till DEBUG? Ja, först måste vi stoppa datorn med STOP-knappen på kontrollpanelen. Sedan sätter vi in DEBUG-adressen (dvs :1EØØ) i P-registret.

När vi nu släcker STOP-lampan med STOP-knappen och trycker på RUN får vi följande svar på TTY:n

```
DBG
>
```

DEBUG har därmed presenterat sig och vi har alltså hamnat rätt i minnet. Observera att DEBUG svarar med "vinkel" (>) medan OMEGA svarar med frågetecken (?).

Innan vi börjar arbeta med DEBUG måste vi kopiera SCRATCHPAD till SCRATCHPAD-KOPIA så att vi senare kan återställa OMEGA:ns adresser i SCRATCHPAD. Med DEBUG gör man en kopiering mycket enkelt. Kommandot lyder.



Så fort vi slagit in punkten svarar DEBUG med ny vinkel och papperet i TTY:n får därefter följande utseende

```
DBG
>CØ FF 1DØØ.
>
```

Vi kan göra kopieringen i DEBUG ändå enklare om vi inför enkla koder för aktuella start- och slutadresser. Följande adresser kan därvid vara av intresse och vi använder som kod bokstaven R följt av en hexadecimal siffra (jämför med fig 3. 93)

R1 = SCRATCHPADKOPIA, startadress (:1DØØ)
R2 = SCRATCHPADKOPIA, slutadress (:1DFF)
R3 = SCRATCHPAD, startadress (:Ø)
RA = OMEGA, startadress (:1ØØ)
RB = LAMBDA, startadress (:1AØØ)
RC = BDP/VER, startadress (:1C4Ø)
RF = Relokeringsbas (:17ØØ)

Nu sätter vi in de förenklade adresskoderna i DEBUG. De kallas register och det hela tillgår med följande enkla kommandon:

```
R11DØØ.
R21DFF.
R3Ø.
RA1ØØ.
RB1AØØ.
RC1C4Ø.
RF17ØØ.
```

Efter varje punkt svarar DEBUG med vinkel och TTY:utskriften ser därför ut på följande sätt

```
DBG
>CØ FF 1DØØ.
>R11DØØ.
>R21DFF.
>R3Ø.
>RA1ØØ.
>RB1AØØ.
>RC1C4Ø.
>RF17ØØ.
>
```

Nu har vi kört in OMEGA och satt upp enkla adresskoder i DEBUG. Inkörningen är därmed avslutad och både OMEGA och DEBUG är därmed klara att användas.

3.4.1.5 Vårt första assemblerprogram

Vi har tidigare i fig 3.89 skrivit ett program i maskinkod. (Ett program för indikering av ASCII-koden på kontrollpanelen). Vi ska nu ställa upp samma program med hjälp av OMEGA.

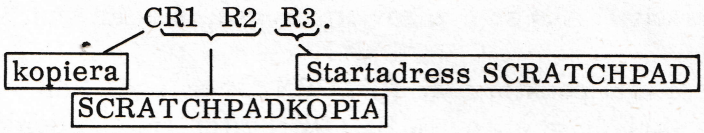
Vi förutsätter nu att minnet i vår ALPHA LSI tidigare laddats med OMEGA, LAMBDA, DEBUG och BDP/VER samt att OMEGA och DEBUG körts in på det sätt som visats i föregående avsnitt.

Eftersom vi skall använda OMEGA måste vi först återställa OMEGA:s SCRATCHPAD, dvs vi måste kopiera SCRATCHPADKOPIA till SCRATCHPAD. Kopiering gör vi enkelt med hjälp av DEBUG.

Om datorn inte varit avstängd sedan arbetet i föregående avsnitt ligger vi i DEBUG. (Vi kontrollerar detta enkelt genom att trycka in punkten på TTY:n. Får vi en vinkel som svar ligger vi i LAMBDA men får vi däremot ett frågetecken som svar så ligger vi i OMEGA).

Om datorn varit frånkopplad så startar vi med P-registret på adressen :1EØØ (dvs på DEBUG:s startadress).

Vi ligger alltså nu i DEBUG och kopierar OMEGA:s SCRATCHPADKOPIA till SCRATCHPAD med följande enkla kommando:



Nu är OMEGA intakt och vi hoppar därför över till OMEGA med kommandot

JRA. ("jump" till adress "RA", dvs till OMEGA)

Utskriften på TTY:n ser ut på följande sätt

```
>CR1 R2 R3.
>JRA.
?
```

Har OMEGA tidigare använts kan det vara lämpligt att nollställa buffern med kommandot I. (initiera) och upprepa "connect"-kommandona (CI5., CO1. och CL1.).

Vi ligger nu i OMEGA. Det är många olika kommandon vi kan använda i OMEGA. (Se appendix A1). Flera av dessa kommandon används för inskrivning och omflyttning av data i den buffer vi tidigare beskrivit.

När vi ligger i OMEGA kan vi direkt skriva in programmet via TTY:n. Programmet hamnar därvid i OMEGA:s buffer. Men observera, nu räcker det med enbart MEMO-symboler!

Vi ger först OMEGA kommandot "AF." som betyder "add keyboard lines to buffer after last line". OMEGA är nu redo att ta emot både instruktioner (i form av MEMO-symboler) och direktiv (dvs kommandon om hur OMEGA skall utföra programmet).

Assemblerinstruktionerna måste skrivas i ett bestämt format. Vi ska inte här gå in på alla detaljer men en sak är väsentlig. När vi nu ska skriva ner våra instruktioner skall vi börja varje ny rad med ett mellanslag (space) och avsluta varje färdigskriven rad med RETURN (CR = carriage return). Vi ska strax förklara varför, men först ska vi skriva ner vårt program. Vi vill ha programmet relokerbart och börjar därför med direktivet

(mellanslag) REL (mellanslag) Ø (RETURN)

Vi fortsätter sedan med MEMO-instruktionerna enligt fig 3. 89.

```
(mellanslag) SEL 7, 4  (RETURN)
(mellanslag) SEL 7, Ø  (RETURN)
(mellanslag) SEN 7, 1  (RETURN)
(mellanslag) JMP $-1   (RETURN)
(mellanslag) RDA 7, 1  (RETURN)
(mellanslag) OCA       (RETURN)
(mellanslag) JMP $-4   (RETURN)
(mellanslag) END       (RETURN)
(RETURN)
```

Sista raden är ett direktiv (END) som anger att programmet är slut.

Observera att vi inte använt punkter utan RETURN för ny rad. När vi skrivit in alla rader i vårt program skriver vi ett extra RETURN efter sista raden och då svarar OMEGA med ett frågetecken.

Utskriften på TTY:n ser ut på följande sätt:

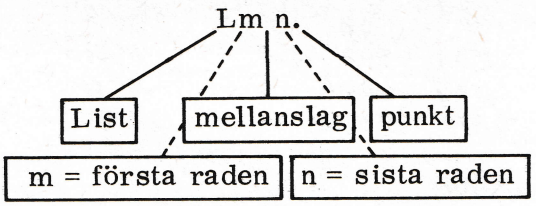
```
SEL    7,4
SEL    7,Ø
SEN    7,1
JMP    $-1
RDA    7,1
OCA
JMP    $-4
END
?
```

Normalt skriver man en mängd fel när man "knackar" in en lista av ovanstående typ på TTY:n. Det finns därför flera kommandon för att korrigera fel eller göra ändringar - men för att inte krångla till tankegången antar vi att vår lista ovan blev rätt inskriven redan vid första försöket. Men hur kontrollerar vi det?

Det gör givetvis OMEGA om vi bara ger rätt kommando. Vi försöker med

L1 F.

som betyder "list buffer lines 1 through F", där F är sista raden i vår lista. I appendix A1 är detta kommando utskrivet som



Vi kan alltså "lista" godtyckliga rader med detta kommando, men vi har här valt att lista samtliga, från början (1) till slut (F).

Så fort vi satt punkt efter kommandot börjar TTY:n rassla och vi får utskriften i fig 3.94. MEMO-symbolerna kallas "källprogram" eller "källkod".

```
?L1 F.
0001   REL 0
0002   SEL 7,4
0003   SEL 7,0
0004   SEN 7,1
0005   JMP $-1
0006   RDA 7,1
0007   OCA
0008   JMP $-4
0009   END
```

Fig 3.94 Källprogram (samma program som i fig 3.89) "listat" från OMEGA:s buffer.

I vänstra kolumnen står radnummer och i den högra instruktioner (och direktiv) skrivna i MEMO-språket. Vårt källprogram har tydligen blivit rätt inskrivet.

Nu kommer vi till det som kallas "assemblering". Vi ger kommandot "X." som betyde "assemble". Genast börjar TTY:n väsnas på nytt och vi får den utskrift som visas i fig 3.95.

```
?X.
PAGE  0001

0001  0000                REL    0
0002  0000  403C          SEL    7,4
0003  0001  4038          SEL    7,0
0004  0002  4939          SEN    7,1
0005  0003  F601          JMP    $-1
0006  0004  5939          RDA    7,1
0007  0005  4404          OCA
0008  0006  F604          JMP    $-4
0009                      END
0000 ERRORS

PUNCH ON, RUN. AT HALT OFF, RUN.
```

Fig 3.95. Källprogrammet i fig 3.94 assemblerat i OMEGA

Överst står sidnummer (PAGE ØØØ1). Utskriften görs i sidor så att man lätt skall kunna förvara den. Vårt program är så kort att vi enbart behöver en del av första sidan.

Efter sidnumret följer en utskrift i fyra kolumner. Till vänster ser vi radnummer precis som i föregående utskrift. Andra kolumnen visar nummer i programmet.

Tredje kolumnen känner vi igen! Det är våra tidigare med mycken möda hopsatta maskininstruktioner. Jämför med fig 3.89!

Fjärde kolumnen innehåller de tidigare inskrivna MEMO-symbolerna.

Sista raden ger oss ett glatt budskap: "ØØØØ ERRORS", dvs inga fel. Assemblern letar alltså fel åt oss och "listar" dem så att vi kan korrigera misstagen.

För att inte tappa tråden har vi här antagit att allt blev rätt redan i första försöket. (Normalt får man göra många korrigeringar vid programmeringsarbete).

Efter felutskriften rusade papperet i TTY:n fram till ny sida. Det verkar slöseri med papper men det gäller ju att ha ordning och reda på sidorna. På ny sida fick vi utskriften

PUNCH ON, RUN. AT HALT OFF, RUN.

Det är bara att lyda och trycka in ON-knappen på hålstansen.

Men inget händer! Ja, datorn har stannat och väntar att vi ska starta den. Vi måste trycka på RUN på kontrollpanelen för att få datorn att arbeta vidare. När TTY:n stansat färdigt trycker vi lydigt på OFF-knappen på hålstansen. Även här har datorn stannat och vi startar den på nytt med RUN-knappen. OMEGA svarar med frågetecken.

Programmet är färdigt. Vi har det i form av en hålremsa skriven i "objekt-kod". Det här var ju bekvämare än att trycka in maskinkoden via kontrollpanelens knappar. Och dessutom har vi nu programmet arkiverat på remsa så att vi kan spara det för framtida bruk.

Om vi vill kan vi även spara "källprogrammet", dvs den lista av MEMO-symboler vi tidigare skrev in i OMEGA:ns buffer. Det går enkelt med följande kommando

PLT 1 F.

vilket betyder "punch the buffer with leader and trailer". OMEGA svarar på känt sätt med

PUNCH ON, RUN. AT HALT OFF, RUN.

Om vi lyder datorn lika bra nu som tidigare, så har vi strax en hålremsa utstansad med källkoden.

Samtidigt får vi motsvarande utskrift på TTY:s teckenskrivare.

```
?PLT 1 F.
PUNCH ON, RUN. AT HALT OFF, RUN.
 REL Ø
 SEL 7,4
 SEL 7,Ø
 SEN 7,1
 JMP $-1
 RDA 7,1
 OCA
 JMP $-4
 END
```

Innan vi lämnar OMEGA ska vi bara förklara det mystiska mellanslaget i början av varje rad när vi tidigare skrev in MEMO-koden. Man kan börja en instruktion med en "label", dvs en etikett som anger adressen till instruktionen.

På rad ØØØ4 står instruktionen SEN 7, 1. Vi kan ge denna instruktion en "label", exempelvis HEJ. Då börjar raden med HEJ (och inte med mellanslag) och instruktionen blir

HEJ SEN 7, 1

När vi sedan kommer till rad ØØØ8 kan vi byta ut instruktionen JMP $-4 mot JUMP HEJ. På detta sätt slipper vi räkna hoppsteg. Risken för fel om vi inlänkar fler instruktioner i programmet minskar. Assemblera detta program som övning! (Den erhållna maskinkoden blir densamma som i fig 3. 95).

OMEGA är alltså en bra hjälpreda när det gäller att hålla reda på adresser.

Vi har nu fått en antydan om hur OMEGA arbetar. Men vi har inte provkört vårt program ännu! Det ska vi göra med hjälp av DEBUG.

3.4.1.6 Källkod, objektkod och binärkod

Vid assembleringen med OMEGA fick vi ut en "objekt-remsa". Den innehåller maskinkoden för vårt program samt dessutom start- och stopptecken, uppgifter om programmets längd samt vissa "typkoder". Flera "objekt-remsor" kan ställas samman till ett större program med hjälp av LAMBDA. Ett sådant program kan exempelvis bestå av ett huvudprogram samt ett antal subrutiner.

LAMBDA lagrar ett sammanhängande program (som alltså inkluderar alla inmatade objekt-remsor) i datorns minne. Detta program kan "dumpas ut" av BDP/VER i form av en "binärremsa". Denna "binärremsa" kan sedan laddas av AUTOLOAD. (För enklare program är binärremsan och objektremsan så likartade att även objektremsan kan läsas in av AUTOLOAD).

Vi ska alltså skilja mellan objekt-remsa (eller objekt-kod) och binärremsa (eller binärkod). Objektremsor skall bearbetas av LAMBDA till färdiga program i form av binärremsor.

Binär-remsor är den normala formen av "software" (som i fig 3.91) och laddas direkt av AUTOLOAD.

Fig 3.96 sammanfattar de olika programpaketens uppgifter vid programmeringsarbetet.

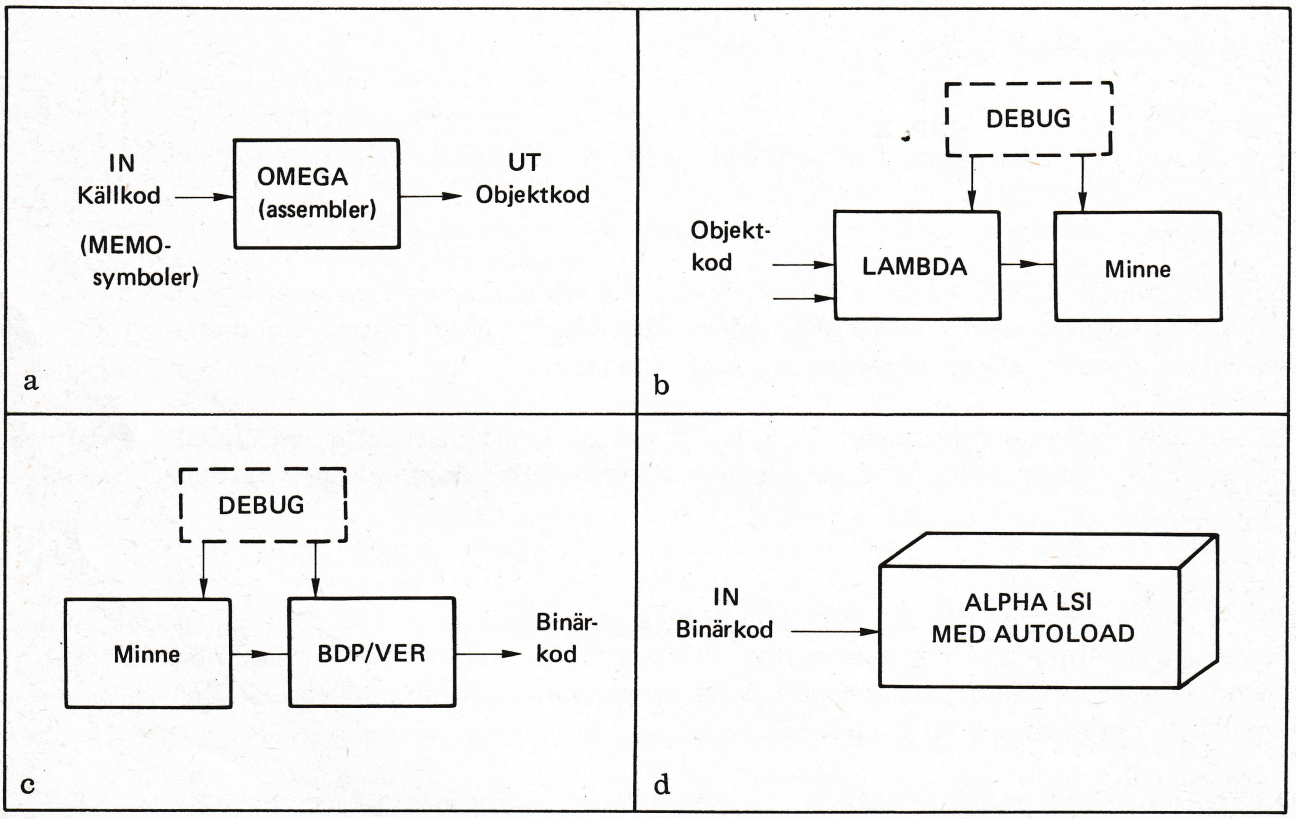


Fig 3.96 Källkod, objektkod och binärkod.

Enligt fig 3.94 skriver vi alltså programmet med MEMO-symboler. Detta program kallas källkod (source program eller källprogram) och assembleras med hjälp av OMEGA. Ut från OMEGA kommer en objektkod (och om vi så önskar även en källkod).

Flera objektkoder kan laddas in med LAMBDA. Programmet kan "avlusas" och provköras med DEBUG och när det fungerar på avsett sätt kan det köras ut via BDP/VER (binary dump) till en sammanhängande binärremsa (binärkod).

Binärremsan laddas med det i datorn inbyggda programmet AUTOLOAD.

3.4.1.7 Från objektremsa till binärremsa

Nu ska vi alltså provköra vårt program från fig 3.95.

Vi börjar med att sätta startadressen för DEBUG (dvs :1EØØ) i P-registret och därefter trycker vi på RUN.

DEBUG presenterar sig på känt sätt på TTY:n med

```
DBG
>
```

Vår avsikt är nu att använda LAMBDA för laddning av den objektremsa vi fått ut från OMEGA efter assembleringen.

I DEBUG kan vi ge kommandon som sätter datorns A-register, X-register och SENSE-register i önskat läge vid ett senare hopp från DEBUG till LAMBDA. I appendix A3 ser vi att det är just dessa register som måste sättas när LAMBDA skall köras. Om LAMBDA körs direkt måste man alltså knappa in dessa värden på kontrollpanelen. När vi nu kör LAMBDA via DEBUG gör vi istället följande kommandon på TTY:n

XØ. (sätt X-reg = Ø)
ARF. (sätt relokeringsadress = RF)
OØ. (sätt SENSE-reg = Ø)

Nu monterar vi den objektremsa vi fick ut vid assembleringen på remsläsaren och ställer omkopplaren på läsaren i läge START.

Med kommandot

JRB.

hoppar vi till adress RB (dvs :1AØØ), dvs till LAMBDA:s startadress. Genast börjar remsan läsas in.

LAMBDA avslutar inläsningen med att låta TTY:n skriva ut första fria positionen i scratchpad samt första fria positionen efter programmet. TTY-utskriften blir följande

```
DBG
>XØ.
>ARF.
>OØ.
>JRB.

E ØØFD 17Ø7
```

Sedan stannar datorn. (Här har vi möjlighet att montera fler remsor och starta med RUN-knappen).

När vår (enda) objektremsa laddats in av LAMBDA går vi tillbaks till DEBUG på samma sätt som tidigare (:1EØØ → P och därefter RUN).

I DEBUG kan vi nu kontrollera om programmet ligger på rätt plats, dvs med startadress RF = 17ØØ. Vi ger PRINT-kommando

P17ØØ 17Ø6.

DEBUG svarar och utskriften på TTY:n får därefter utseendet

```
DBG
>P17ØØ 17Ø6.
17ØØ 4Ø3C 4Ø38 4939 F6Ø1 5939 44Ø4 F6Ø4
```

Vårt program ligger alltså på önskad plats. Låt oss nu köra det!

Med ett hoppkommando

JRF.

lämnar vi DEBUG och hamnar på programmets startadress (:17ØØ). Nu exekveras vårt program av ALPHA LSI.

Tryck på en tangent på TTY:n och kontrollera vad som händer på kontrollpanelens indikator. (S REG/DATA-lampan ska vara släckt).

Programmet har tydligen utfallit till belåtenhet (hoppas vi)!

Vi återgår nu till DEBUG på känt sätt (:1EØØ → P och RUN). Det är nu dags att ta ut programmet i form av en binärremsa.

BDP/VER kräver viss inställning av A-, X- och SENSE-register (se Appendix A3). Vi ger därför DEBUG följande kommandon

OØ. (remsa stansas på TTY)
A17ØØ. (startadress)
X17Ø7. (slutadress)

Därmed har vi meddelat BDP start- och slutadress på de celler som skall dumpas ur minnet.

Vi trycker nu in stansens ON-knapp och ger sedan hoppadress till dump-programmet

JRC.

BDP/VER svarar med en binärremsa från stansen och ett antal obegripliga tecken på teckenskrivaren. (Teckenskrivaren förstår ju inte binära tecken utan enbart ASCII-kod.)

Vårt programmeringsarbete är nu slutfört och vårt program är dokumenterat i en binärremsa.

När vi vid ett senare tillfälle vill använda programmet kan vi ladda ALPHA LSI med vår binärremsa med hjälp av AUTOLOAD. Vi använder då rutinen 1-10 i avsnitt 3.4.1.1.

3.4.1.8 Felsökning med DEBUG

DEBUG är ett mycket effektivt hjälpmedel för felsökning i program. Det kan ske bl a med hjälp av brytpunkter (breakpoints) som placeras i programmet. När programmet körs med DEBUG stoppas exekveringen så fort en brytpunkt uppträder och innehållet i datorns A-, X- och O-register skrivs ut. På detta sätt kan man följa datorns arbete i detalj och lättare upptäcka var eventuella fel i programmet ligger gömda.

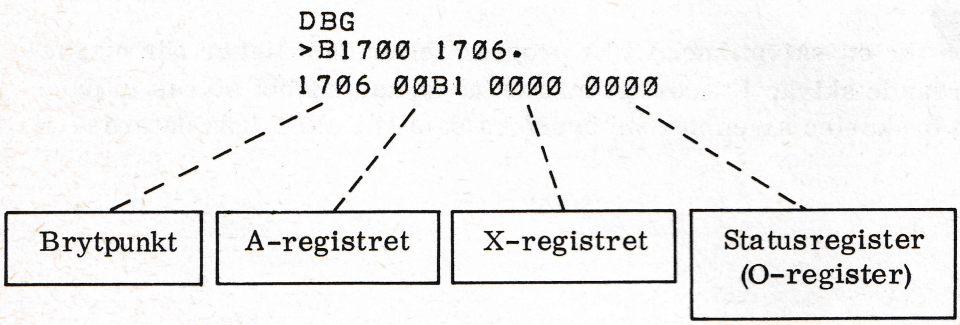
Vi ska nu sätta in en brytpunkt i vårt tidigare program och visa principen.

Vi lägger oss alltså i DEBUG på känt sätt (:1EØØ → P och RUN) och ger därefter kommandot

B17ØØ 17Ø6.

B betyder brytpunkt och DEBUG exekverar nu det program som ligger i minnets positioner :17ØØ till :17Ø6. Position :17Ø3 innehåller en vänta-instruktion (JMP $-1) och så fort vi tryckt ner en tangent fortsätter exekveringen med instruktionen i position :17Ø4. Låt oss trycka ner siffran 1 på TTY:n. ASCII-koden för talet 1 är :B1. Programmet exekveras nu fram till position :17Ø6. Här stoppar DEBUG exekveringen och utskriften på TTY:n får följande utseende

```
DBG
>B17ØØ 17Ø6.
17Ø6 ØØB1 ØØØØ ØØØØ
```



Vi ser här att A-registret innehåller ASCII-koden för siffran 1. I vårt program används varken X- eller statusregistren och de är därför nollställda.

På detta sätt kan man "bit för bit" med hjälp av ett antal brytpunkter söka sig igenom ett stort program och successivt följa upp hur det fungerar.

3.4.2 HÖGNIVÅSPRÅK

De flesta datoranvändare kommer varken i kontakt med maskinprogrammering eller assemblerprogrammering av en dator. De använder sig istället av mera lättarbetade programspråk som FORTRAN (Formula translation program) eller BASIC (Beginners all purpose symbolic instruction code).

Programmering med dessa högnivåspråk ligger helt utom ramen för denna bok. Vi ska enbart nämna något om möjligheterna att använda dessa språk på ALPHA LSI.

Högnivåspråk kräver stort primärminne. Det finns flera BASIC-tolkar, dvs programpaket för körning av BASIC, till ALPHA LSI.

BASIC-2 kräver 8 kord minne och BASIC-4 kräver 16 kord minne. Den senare ger större precision vid beräkningar och har filhantering.

BASIC-tolkarna finns tillgängliga i form av hålremsor som laddas med AUTOLOAD.

FORTRAN kräver operativsystem och 16 kord minne, samt skivminne.

FORTRAN-kompilatorn ligger tillsammans med en rad andra program på den systemskiva som finns tillgänglig för ALPHA LSI:s operativsystem med disk.

3.4.3 OPERATIVSYSTEM

Utvecklingen går mot allt kraftfullare programsystem. Om man snabbt och enkelt vill ha access till en mängd olika program är det opraktiskt och tidsödande att ladda hålremsor.

Om man skaffar ett skivminne ("disk") får men snabb access till avsevärda minnesareor (exempelvis 10M byte). För att hålla reda på programlagringen på skivan och för hanteringen med att "lyfta in" önskade program i primärminnet har datorfabrikanterna utvecklat operativsystem (OS). Tack vare dessa operativsystem kan man med enkla och överskådliga kommandon snabbt och effektivt hantera program och data.

Fig 3.97 visar principen för ett skivminne. Olika program eller data ligger här magnetiskt lagrade på en snurrande skiva. Ett rörligt magnethuvud kan snabbt styras in på önskat "spår" och inom bråkdelen av en sekund överföra data till eller från datorns primärminne.

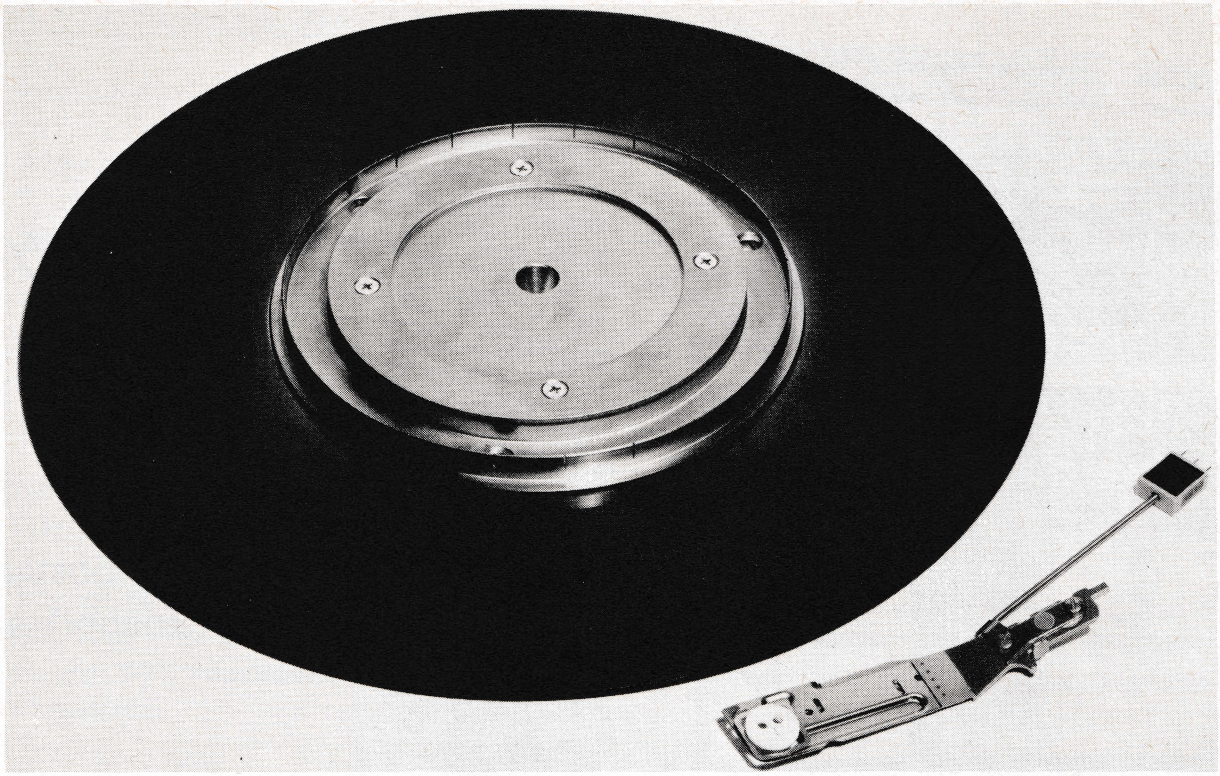


Fig 3.97 Principen för en skiva.

Fig 3.98 visar ALPHA LSI med skivminne, snabbläsare och snabbstans.

Om man ej har så stora krav på snabbhet kan operativsystemet köras på magnetband (kassett eller hjul).

Till ALPHA LSI finns ett väl utvecklat operativsystem som bl a innehåller programmen

OS:ASM,	Assembler
MACRO2,	Avancerad Assembler, s k makroassembler
OS:LDR,	motsvarar LAMBDA
OS:DBG,	motsvarar DEBUG
OS:DMP,	motsvarar DMP/VER
OS:CPY,	kopieringsprogram
OS:VEW,	utskrift av operativsystemets innehållsregister
OS:EDT,	editering och textbehandling

Dessutom finns program för körning av ett flertal högnivåspråk via operativsystemet. Exempel härpå är FORT:4 (FORTRAN) och BASIC4 (en av flera BASIC-varianter).

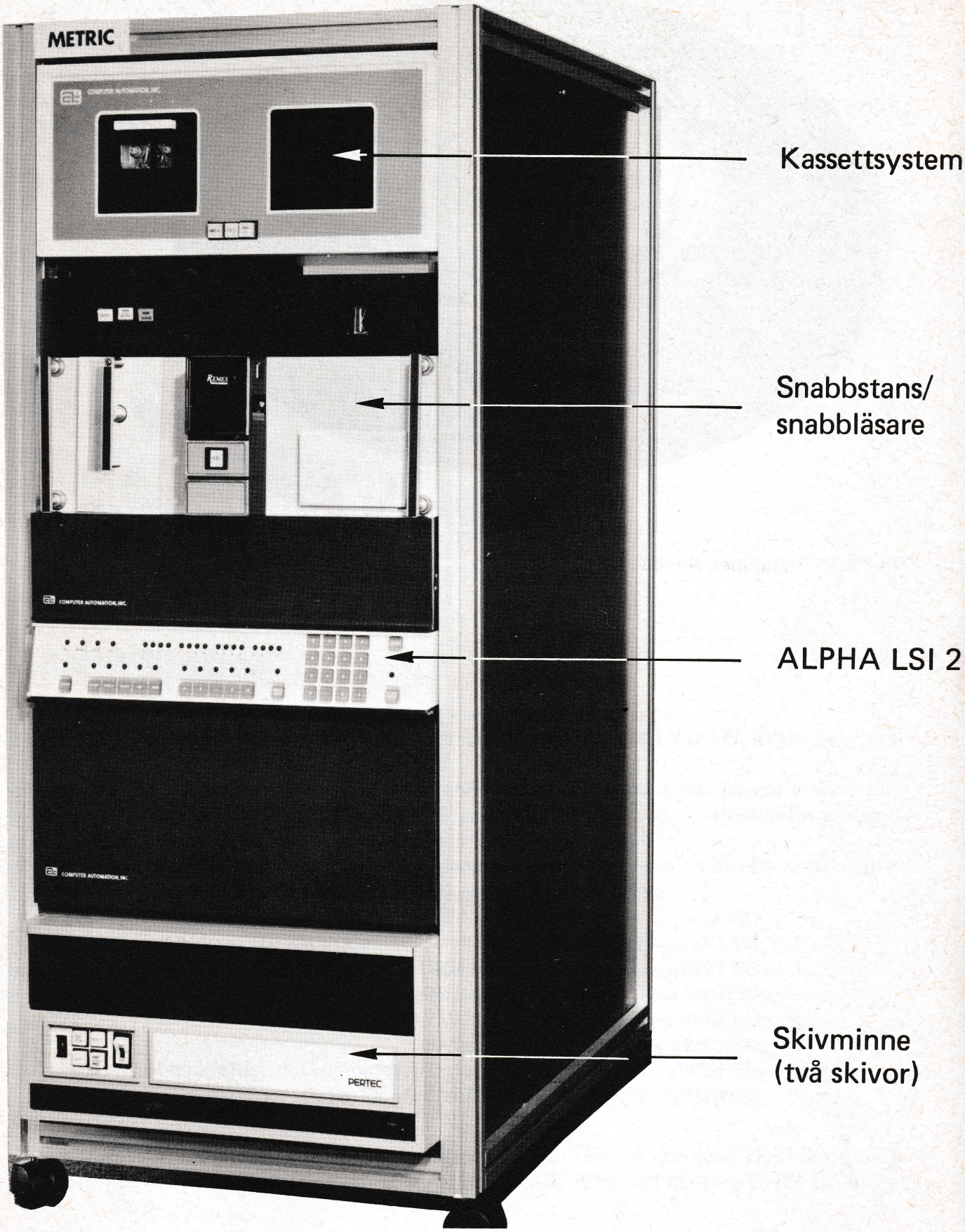


Fig 3. 98 ALPHA LSI med skiva (disk), kassettsystem, snabbstans och snabbläsare.

3.5 KASSAPROGRAMMET - ett avslutande exempel

I de tidigare avsnitten har vi lärt känna

- hur datorn exekverar ett enkelt program
- hur de elektroniska kretsarna i princip fungerar - och speciellt hur styrenheten arbetar,
- hur TTY:n fungerar som I/O-enhet
- och hur det finns hjälpmedel (exempelvis en assembler) för sammanställning av program.

Vi har därmed nått det mål vi inledningsvis ställt upp. Vi har fått ett sammanhang mellan funktionen hos de elektroniska kretsarna (maskinvaran) och hur dessa kretsars funktion kan styras av ett lämpligt utformat program (programvaran).

Det är som tidigare nämnts karakteristiskt för modern elektronik att allt fler funktioner blir programstyrda. Programvaran får därmed en ökande andel av totalkostnaden.

För att få en verklighetsnära bild av både programvara och användning av en minidator ska vi nu som avslutning bekanta oss med "kassaprogrammet". Med dess hjälp ska vi köra vår TTY som kassaterminal och hålla reda på lager och kundkonton i en självbetjäningsaffär.

Det är principen vi vill visa och därför är vår affär givetvis högst begränsad både när det gäller antalet varuslag och kunder.

Den läsare som vill spara kassaprogrammet till ett senare tillfälle kan lämpligen hoppa till avsnitt 4.

3.5.1 ÖNSKAD FUNKTION

Vi vill använda vår TTY som kassaterminal i en affär som säljer följande fem varuslag.

Vara A med priset	5 kr /st
Vara B med priset	15 kr/st
Vara C med priset	25 kr/st
Vara D med priset	3 kr/st
Vara E med priset	9 kr/st

Affären har 30 kunder som alla har konton (dvs köper på kredit) med kontonumren 1 - 30.

Självbetjäningen tillgår på följande sätt:

- På terminalens förfrågan slår kunden in sitt kundnummer. Som avslutning på sitt svar slår kunden in RETURN-tangenten ("termineringskaraktär").
- Terminalen ber nu kunden ange antal och varuslag och kunden svarar med att beställa önskat antal (maximerat till 9) av en viss vara (endast ett varuslag per beställning). Datorn förväntar sig svaret i form av tre nedslag på TTY:n. Det första skall vara en siffra (1-9), det andra en bokstav (A-E) och det tredje RETURN.

- Datorn kollar om varan finns i lager (i önskat antal) och ger sedan utskrift om leverans. Utskriften på TTY:n kan därefter se ut på följande sätt:

```
KUNDNUMMER (1-30,F,L ELLER R)?
1
DIN KONTOSTAELLNING AER  KR:+00000
ANTAL OCH VARUSLAG (NX: N=1-9, X=A-E)?
9A
+00009 ST KAN HAEMTAS VID LASTKAJEN!
```

Om önskat antal överstiger lagrets innehåll får vi exempelvis svaret:

```
BESTAELLD MAENGD FINNS EJ I LAGER!
+00007 ST KAN HAEMTAS VID LASTKAJEN!
```

Angivet antal är härvid allt som återstår i lagret vid vår beställning.

Om varan är slut i lagret får vi svaret

```
SLUT I LAGER.
```

Affärsföreståndaren skall ha följande rutiner till hjälp:

- På kommandot L (lagerställning) skall datorn skriva en lagerlista så att affärsföreståndaren kan göra beställning och fylla lagret.
  Fullt lager är i vårt fall 30 st varor av varje slag. Efter en tid kan lagerställningen se ut på följande sätt:

```
L
LAGERSTAELLNING:
A=+00000
B=+00024
C=+00030
D=+00030
E=+00030
```

Affärsinnehavaren kan då göra beställning hos grossisten och fylla sitt lager.

- När lagret blivit påfyllt ger föreståndaren meddelandet F (lagret fyllt) till datorn.
- Vid lämpliga tidpunkter ger föreståndaren datorn order om utskrift av räkningar (fakturor). Kommandot härför är R.

En faktura kan ha följande utseende:

```
FAKTURA:

KUNDNUMMER +00002      ATT BETALA KR:+00135
```

Man frågar sig nu hur ett program ska se ut om det ska kunna fylla alla dessa önskmål?

Svaret på den frågan är inte entydigt. Man kan göra ett (nästan) obegränsat antal program som fyller de uppställda önskemålen. Vi ska nu presentera ett exempel på lösning.

Har vi tillgång till en ALPHA LSI så börjar vi med att skaffa binärremsan till KASSA-PROGRAMMET. Vi laddar sedan in det med AUTOLOAD och kör (starta med :1ØØ → P och RUN). Vi måste vara medvetna om att programmet förstör tidigare inmatade program, exempelvis OMEGA.

När programmet startats kan vi prova med några olika beställningar och med kommandona L, F och R.

Allt tycks fungera enligt önskemålen och därmed har vi all anledning att vara nyfikna på programmets uppbyggnad.

Vi ska nedan diskutera programmets utformning i två steg. Först en allmän blockbeskrivning som man i programmeringssammanhang kallar "flödesplan". Därefter ska vi bekanta oss med några detaljer i assemblerprogrammet.

Som tidigare nämnts är detta ingen lärobok i programmering och vi har inte utrymme för att beskriva alla intressanta detaljer i kassaprogrammet. Vi hoppas emellertid att följande avsnitt dels ska ge en bra översikt över ett programs uppbyggnad och dels motivera fortsatta studier i assemblerprogrammeringens sköna konst.

För alla som i fortsättningen kommer att syssla med elektroniska utrustningar i anslutning till mini- eller mikrodatorer är assemblerprogrammering ett väsentligt område. Visserligen har varje datorfamilj sitt speciella assemblerspråk (dvs sin speciella uppsättning instruktioner) men arbetssättet vid assemblerprogrammering är i princip detsamma för alla mini- och mikroprocessorer.

3.5.2 FLÖDESPLANEN

När man gör ett program brukar man börja med en flödesplan som i stort anger hur programmets tilltänkta funktion kan uppdelas i ett antal deloperationer. En flödesplan för kassaprogrammet visas i fig 3.99. Man brukar använda speciella blocksymboler för flödesplaner. Fig 3.100 visar exempel på hur dessa blocksymboler ritas.

En flödesplan är som en disposition för en uppsats. Den kan skrivas i några få huvudrubriker eller mycket detaljerat med många underrubriker.

Kassaprogrammets flödesplan i fig 3.99 är av praktiska skäl uppdelad i två delar men de hänger i själva verket ihop. Uthoppet ② från den vänstra sker till inhoppet ② i den högra. Dessutom sker uthopp till ① från ett flertal blocksymboler.

Låt oss till en början bortse från alla siffror som anges vid sidan av blocken i flödesplanen. De hör i själva verket inte till planen utan hänvisar till radnummer i det assemblerprogram vi senare skall detaljstudera.

Vad säger oss flödesplanen?

Vi kan följa blocken uppifrån och nedåt:

- Efter start fylls lagret med hjälp av en subrutin. (FYLL LAGER).
- TTY:n skriver "Kundnummer (1-30, F, L eller R)?"
- Kunden svarar med sitt kundnummer (IN SVAR).
- Datorn kontrollerar svarets längd med hjälp av en subrutin. I alla program spelar olika typer av kontroller en viktig funktion. Slår en kund in för långt nummer kan programmet förstöras om inte en rad säkerhetsåtgärder vidtagits. (Vi antar för tillfället att svaret ej är för långt).
- Datorn jämför svaret med F, L och R. Om svaret är F, L eller R vidtas lämpliga åtgärder med hjälp av tre subrutiner, Lagerställning, Fyll lager, Räkning och nollställning. När dessa subrutiner exekverats sker uthopp till①, dvs till ny utskrift av "Kundnummer. (1-30, F, L eller R)?" på TTY:n.
- Om datorn ej kan identifiera svaret F, L eller R prövar den om svaret är något av talen 1-30. Om så är fallet avser talet ett kundnummer och denna kunds kontoställning skrivs ut. I annat fall skrivs ett felmeddelande ut och därefter ges ny förfrågan om kundnummer.
- Efter utskriften av kundens kontoställning sker uthopp②, dvs till högra delen av flödesplanen. Datorn börjar här med att fråga efter antal och varuslag.
- När svaret kommit in kollas dess längd av en subrutin. (Vi antar att svaret ej är för långt).
- Nu följer ytterligare en kontroll om svaret håller sig inom givna gränser, dvs 1-9 och A-E. I annat fall kommer felmeddelande och uppmaning till kunden att skriva beställningen korrekt.
- När korrekt beställning har mottagits kollar datorn lagerställningen. Om varan är slut i lager kommer utskrift därom.
- Om varan finns i lager beräknar datorn differensen mellan beställning och lagerinnehåll. Om beställningen är större än lagret meddelar datorn att beställd mängd ej finns i lagret. Datorn anger därefter att beställt (eller tillgängligt) antal varor kan avhämtas vid lastkajen.
- Slutligen revideras lagerställning (med hänsyn till antal varor) och kundkonton (med hänsyn till antal varor och pris).

Kassaprogrammet ger oss alltså smakprov på en hel del intressanta och nyttiga operationer.

Testa programmet ännu en gång genom att köra programmet och kontrollera om det följer flödesplanen!

Då är vi mogna för detaljerna!

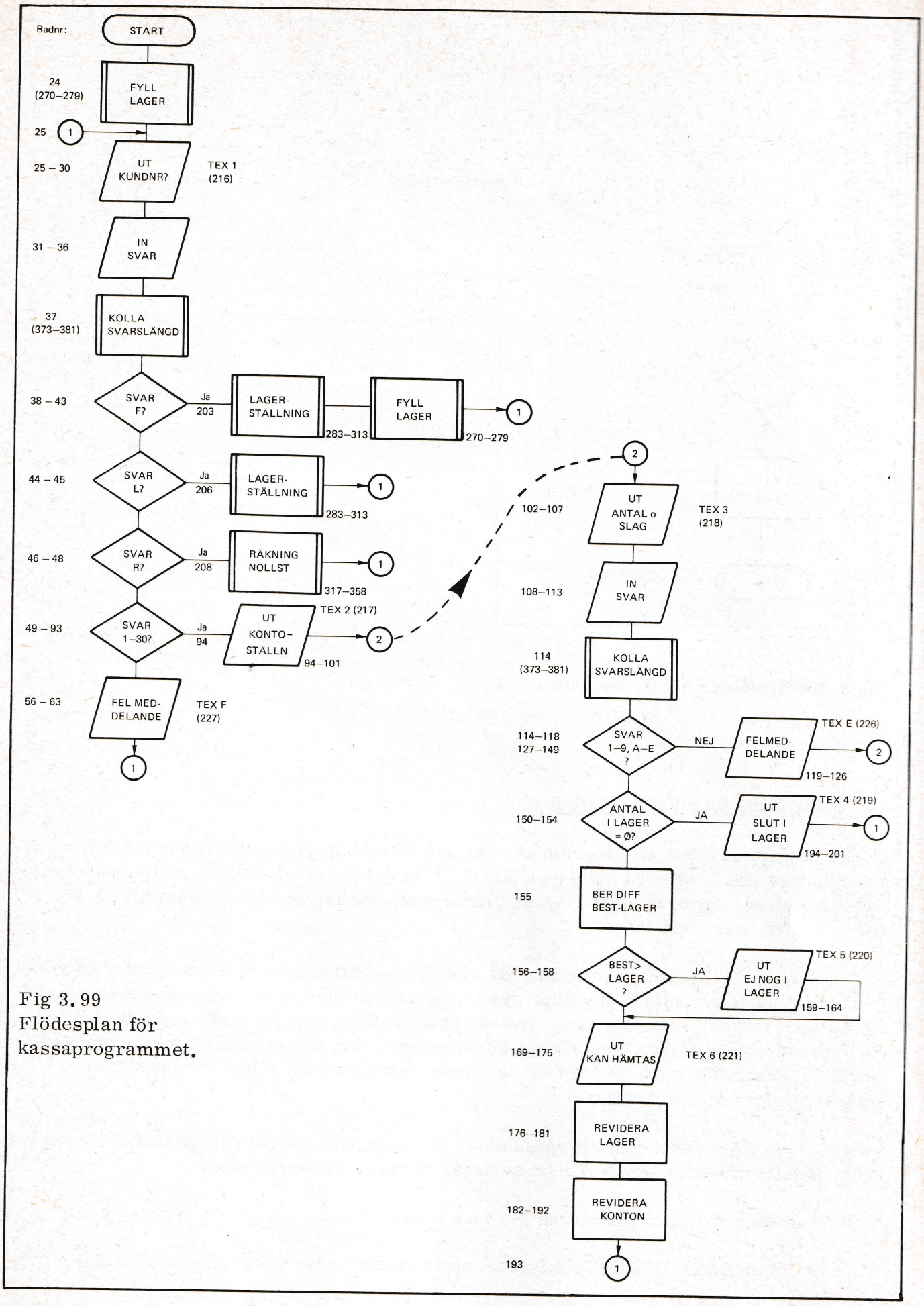


Fig 3.99
Flödesplan för kassaprogrammet.

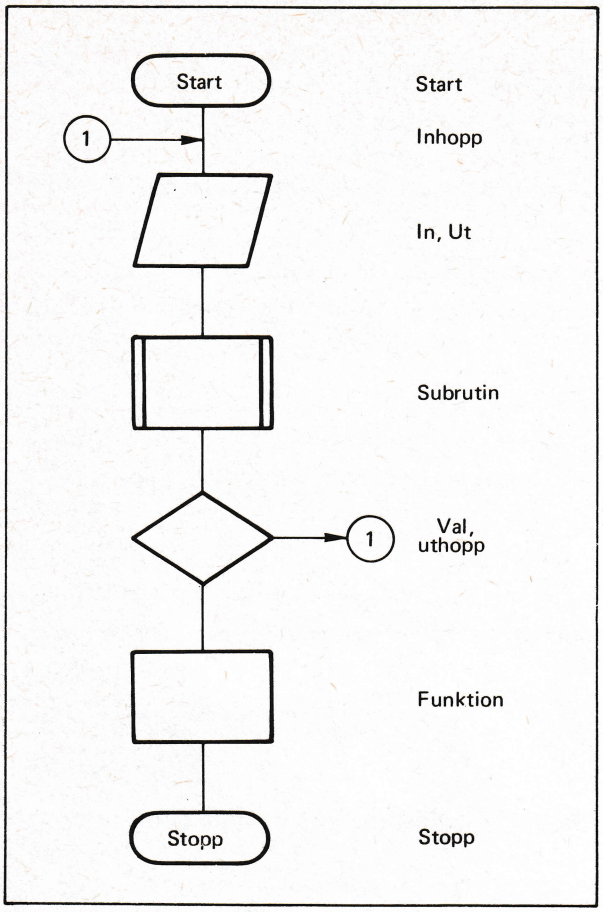


Fig 3.100 Symboler för flödesplanen.

3.5.3 ASSEMBLERPROGRAMMET

Vi har tidigare sett hur ett program kan skrivas med hjälp av MEMO-symboler och assembleras (fig 3.95) med hjälp av OMEGA. I fig 3.101 ser vi utskriften efter assembleringen av kassaprogrammet. Det är i princip samma typ av utskrift som i fig 3.95 men avsevärt mer omfattande.

När vi skriver ett assemblerprogram har vi möjlighet att skriva in kommentarer på varje rad. Dessa kommentarer inverkar ej på programmet utan är enbart avsedda som en förklarande text för dem som skall använda programmet. Som framgår av fig 3.101 är vårt kassaprogram försett med rikliga kommentarer som visserligen på många ställen gjorts kryptiska (för att få plats) men som trots detta ger oss en bra ledning när vi studerar programmets detaljer.

Vi refererar i fortsättningen till radnummer (dvs kolumnen längst till vänster). Första sidan omfattar radnummer 1-57 men av dessa upptas 1-23 av rubriktext.

Huvudprogrammet ligger på sidorna 1-4 (dvs raderna 24-209)

Därefter följer DATA DEFINITIONS, som upptar sidorna 5-7 (dvs raderna 216-261).

Av dessa DATA DEFINITIONS utgörs merparten av text (textsträngar) som är lagrade på raderna 216-230. Varje minnescell om 16 bitar innehåller här två bokstäver lagrade i ASCII-kod. På rad 230 innehåller exempelvis den första cellen (programsteg 0163) hextalet :DØD2. I fig 3.78c ser vi att ASCII-koden för bokstaven P är :DØ och för R är :D2. Bokstäverna lagras alltså i tur och ordning med den första i ordets vänstra BYTE (most significant byte, MSB) och den andra i ordets högra BYTE (least significant byte, LSB).

Rad 231-240 innehåller lagrade tal (MEMO-symbolen DATA är ett direktiv till assemblern att tolka innehållet som ett tal). Enklast att känna igen är här raderna 236-240 som i decimal form anger priserna på varorna A-E. Rad 238 innehåller priset 25 som av datorn skrivs hexadecimalt som ØØ19 i tredje kolumnen.

MEMO-symbolen RES ger assemblern direktiv om att reservera plats i minnet. Rad 254 innehåller RES 3Ø,Ø, vilket betyder "reservera 30 celler, alla skall vara nollställda i utgångsläget". Vi ser i andra kolumnen (programräknarens ställning) att det samtidigt sker ett hopp från Ø18B till Ø1A9 på efterföljande rad (255).

Raderna 258-261 innehåller referenser (adresspekare) till yttre subrutiner. När programmet laddas bestämmer LAMBDA var dessa subrutiner skall placeras. De subrutiner som här utnyttjas ingår i programpaketet TUP (teletype utility package) som levereras med ALPHA LSI.

På sida 8 följer så SUBRUTINER och vi återfinner här följande för kassaprogrammet specialgjorda subrutiner:

FYLLA	(rad 270-279),	fyller lager
LAGST	(rad 283-313),	ger lagerställning
RAKN	(rad 317-358),	ger fakturor
RADF	(rad 362-369),	ger 15 radframmatningar.
CHBFRL	(rad 373-381),	kollar att svaret begränsas till 8 nedtryck. I annat fall följer felutskrift och datorn stoppas (rad 382-393).

Assemblerprogrammet avslutas med en listning av alla etiketter (labels) på sida 11. Alla dessa namn är påhittade av programmeraren och hennes avsikt har varit att (med maximalt sex tecken) antyda funktionen och därmed underlätta läsningen av programmet.

Det är många nya instruktioner vi möter i femte kolumnen och vi har därför sammanställt dessa i fig 3.102. Utöver rena instruktioner har vi även mött vissa direktiv, exempelvis DATA och RES. Direktiven i kassaprogrammet finns sammanställda i fig 3.103.

Vi har nu fått en viss översikt och struktur av assemblerprogrammet. Det kan nu vara dags att detaljstudera några intressanta delar av programmet.

```
PAGE  0001
0001                *************************************************
0002                *
0003                *  KASSAPROGRAMMET
0004                *
0005                *  FROEN KASSA-TERMINALEN KAN BESTAELLAS
0006                *  ANTAL:                 1-9
0007                *  VARUSLAG:              A,B,C,D OCH E
0008                *  GILTIGA KUNDNUMMER :   1-30
0009                *                         L   GER LAGERSTAELLNIG
0010                *                         F   GER LAGERSTAELLNING OCH
0011                *                             FYLLER LAGRET
0012                *                         R   SKRIVER UT FAKTURA OCH
0013                *                             NOLLSTAELLER ALLA KONTON
0014                *
0015                *  PRIS:  VARA A          5  KR
0016                *             B          15 KR
0017                *             C          25 KR
0018                *             D          3  KR
0019                *             E          9  KR
0020                *
0021                *  TERMINERINGS-KAR:      CARRIGE RETURN
0022                *
0023                *************************************************
0024  0000  F900          JST  FYLLA     SUBRUTIN SOM FYLLER LAGRET
            01B4
0025  0001  0F00  KNR     SWM            SET WORD MODE
0026  0002  F900          JST  *CRLF     SUB: CARRIGE RETURN OCH LINE FEED
            81B3
0027  0003  C6BF          LAP  :BF       ASCII-KOD FROEGETKN;TERM-KAR
0028  0004  0F00          SWM
0029  0005  F900          JST  *OTL      SUB: OUTPUT RAD TILL TTY
            81B1
0030  0006  00BC          DATA TEX1      'KUNDNUMMER...?' UT
0031  0007  0F00          SWM
0032  0008  F900          JST  *CRLF
            81B3
0033  0009  C68D          LAP  :8D       TERM-KAR=CARRIGE RETURN
0034  000A  0F00          SWM
0035  000B  F900          JST  *IKL      SUB: INPUT RAD FROEN KEYBOARD
            81B2
0036  000C  017E          DATA BUFFIN    ADRESS TILL BUFFR
0037  000D  F900          JST  CHBFRL    SUB: KOLL OM BUFF-L OEVERSKR.
            020F
0038  000E  0F00          SWM
0039  000F  F900          JST  *CRLF
            81B3
0040  0010  B100          LDA  BUFFIN    SVAR TILL A-REG
            017E
0041  0011  13D7          LRA  8         SKIFTA 8 STEG HOEGER
0042  0012  C0C6          CAI  'F'       SVAR=F?
0043  0013  F2A1          JMP  F         JA,HOPPA TILL F
0044  0014  C0CC          CAI  'L'       NEJ; SVAR=L?
0045  0015  F2A2          JMP  L         JA,HOPPA TILL L
0046  0016  C0D2          CAI  'R'       NEJ; SVAR =R?
0047  0017  F2A2          JMP  R         JA,HOPPA TILL R
0048  0018  B100          LDA  BUFFIN    NEJ; SVAR=1-30?
            017E
0049  0019  8100          AND  HFF       MASKA UT BYTE 2
            0172
0050  001A  C08D          CAI  :8D       BYTE 2=CARRIGE RETURN?
0051  001B  F224          JMP  CH1       OM BYTE 2 VAR CARRIGE RETURN
0052  001C  B100          LDA  BUFFIN+1  ORD 2 AV SVARET TILL A-REG
            017F
0053  001D  13D7          LRA  8         BYTE NR 3 IN
0054  001E  C08D          CAI  :8D       BYTE 3=CR?
0055  001F  F209          JMP  CHKNR     JA,HOPPA TILL FORTSAETTNING
0056  0020  0F00  ERRF    SWM            NEJ; DVS. FELAKTIGT KUNDNUMMER
0057  0021  F900          JST  *CRLF
            81B3
```

```
PAGE  0002
0058  0022  C6A1          LAP  '!'       TERM-KAR
0059  0023  0F00          SWM
0060  0024  F900          JST  *OTL
            81B1
0061  0025  0141          DATA TEXF      'FELAKTIGT KUNDNUMMER...!' UT
0062  0026  0F00          SWM
0063  0027  F900          JST  *CRLF
            81B3
0064  0028  F627          JMP  KNR       HOPPA TILL 'KUNDNUMMER?' UT
0065  0029  B100  CHKNR   LDA  BUFFIN     SVARET TILL A-REG
            017E
0066  002A  8100          AND  HFF       MASKA UT ENTALSIFFRAN UR SVARET
            0172
0067  002B  0DB9          SAI  :B9       ASCII-KOD FOER 9
0068  002C  31CC          JAG  ERRF      KOLL OM SIFFRA>9
0069  002D  0B09          AAI  9         JUSTERA OCH
0070  002E  20CE          JAM  ERRF      KOLL OM SIFFRA<0
0071  002F  9900          STA  ENTAL     LAGRA ENTALSSIFFRAN
            0186
0072  0030  B100          LDA  BUFFIN    SVAR TILL A-REG
            017E
0073  0031  13D7          LRA  8         SKIFTA 8 STEG HOEGER
0074  0032  0DB3          SAI  :B3       ASCII-KOD FOER 3
0075  0033  31D3          JAG  ERRF      KOLL OM SIFFRA>3
0076  0034  0B03          AAI  3         JUSTERA OCH
0077  0035  21D5          JAL  ERRF      KOLL OM SIFFRA <=0
0078  0036  0048          TAX            FLYTTA A-REG TILL X-REG
0079  0037  B100          LDA  ENTAL     MULTIPL TIOTALSSIFFR (X-REG)
            0186
0080  0038  1960          MPY  TIO       MED 10&ADDERA ENTAL(A-REG)
      0039  0173
0081  003A  13A8          LRX  1         JUSTERA SVARET
0082  003B  E900          STX  KUNDN     LAGRA KUNDNUMRET
            0185
0083  003C  0030          TXA            X-REG TILL A-REG
0084  003D  0D1E          SAI  30        KOLL OM KUNDNR>30
0085  003E  31DE          JAG  ERRF      JA,HOPPA TILL ERRF
0086  003F  F207          JMP  KONS      NEJ,HOPPA TILL KONS
0087  0040  B100  CH1     LDA  BUFFIN    SVAR ENDAST EN KARAKTAER
            017E
0088  0041  13D7          LRA  8         SKIFTA 8 STEG HOEGER
0089  0042  0DB9          SAI  :B9       ASCII-KOD FOER 9
0090  0043  31E3          JAG  ERRF      KOLL OM SIFFRA>9
0091  0044  0B09          AAI  9         JUSTERA OCH
0092  0045  21E5          JAL  ERRF      KOLL OM SIFFRA<=0
0093  0046  9900          STA  KUNDN     LAGRA KUNDNUMMMER
            0185
0094  0047  C6BA  KONS    LAP  ':'       TERM-KAR
0095  0048  0F00          SWM
0096  0049  F900          JST  *OTL
            81B1
0097  004A  00CB          DATA TEX2      'DIN KONTOSTAELLNING AER:' UT
0098  004B  E100          LDX  KUNDN     KUNDNUMMER TILL X-REG SOM INDEX
            0185
0099  004C  B500          LDA  @PKONTO   KONTOST.TILL A-REG
            018A
0100  004D  0F00          SWM            SUB: OUTPUT DECIMALTAL
0101  004E  F900          JST  *ODEC     KONTOSTAELLNING UT
            81B0
0102  004F  0F00  BESTIN  SWM
0103  0050  F900          JST  *CRLF
            81B3
0104  0051  C6BF          LAP  :BF       TERM-KAR=?
0105  0052  0F00          SWM
0106  0053  F900          JST  *OTL
            81B1
0107  0054  00D9          DATA TEX3      'ANTAL OCH VARUSLAG...?' UT
0108  0055  0F00          SWM
0109  0056  F900          JST  *CRLF
```

Fig 3.101 a.

```
PAGE  0003
            81B3
0110  0057  C68D          LAP  :8D       TERM-KAR=CARRIGE RETURN
0111  0058  0F00          SWM
0112  0059  F900          JST  *IKL      SUB: INPUT RAD FROEN KEYBOARD
            81B2
0113  005A  017E          DATA BUFFIN    ADRESS TILL BUFFER
0114  005B  F900          JST  CHBFRL    SUB: KOLL OM BUFL. OEVERSKR.
            020F
0115  005C  B100          LDA  BUFFIN+1  ORD 2 AV SVARET TILL AREG
            017F
0116  005D  13D7          LRA  8         BYTE 3 IN
0117  005E  C08D          CAI  :8D       KONTROLLERA OM 3:E KAR=CR
0118  005F  F209          JMP  CHBEST    JA HOPPA TILL CHBEST
0119  0060  0F00  ERRE    SWM            NEJ, DVS FELAKTIG BESTAELLNING
0120  0061  F900          JST  *CRLF
            81B3
0121  0062  C6A1          LAP  '!'       TERM-KAR
0122  0063  0F00          SWM
0123  0064  F900          JST  *OTL
            81B1
0124  0065  012E          DATA TEXE      'FELAKTIG BESTAELLNING...!' UT
0125  0066  0F00          SWM
0126  0067  F900          JST  *CRLF
            81B3
0127  0068  F619          JMP  BESTIN
0128  0069  0F00  CHBEST  SWM            BESTAELLNINGEN OMFATT. 2 KAR
0129  006A  F900          JST  *CRLF
            81B3
0130  006B  B100          LDA  BUFFIN    BEST TILL A-REG
            017E
0131  006C  13D7          LRA  8         ANTALET VAROR IN
0132  006D  0DB9          SAI  :B9       ASCII-KOD FOER 9
0133  006E  31CE          JAG  ERRE      KOLL OM ANT BEST VAROR>9
0134  006F  0B09          AAI  9         JUSTERA OCH
0135  0070  21D0          JAL  ERRE      KOLL OM ANT BEST VAROR<=0
0136  0071  9900          STA  BESANT    ANTAL BEST VAROR
            0184
0137  0072  B100          LDA  BUFFIN    BESTAELLNING TILL A-REG
            017E
0138  0073  82FE          AND  HFF       MASKA UT VARUSLAG
0139  0074  C0C1          CAI  'A'       BESTAELLD VARA=A?
0140  0075  F209          JMP  INDEX     JA,HOPPA TILL INDEX
0141  0076  C0C2          CAI  'B'       NEJ, SVAR=B?
0142  0077  F207          JMP  INDEX     JA,HOPPA TILL INDEX
0143  0078  C0C3          CAI  'C'       NEJ, SVAR=C
0144  0079  F205          JMP  INDEX     JA,HOPPA TILL INDEX
0145  007A  C0C4          CAI  'D'       NEJ, SVAR=D?
0146  007B  F203          JMP  INDEX     JA,HOPPA TILL INDEX
0147  007C  C0C5          CAI  'E'       NEJ, SVAR=E?
0148  007D  F201          JMP  INDEX     JA,HOPPA TILL INDEX
0149  007E  F61E          JMP  ERRE      FELAKTIGT VARUSLAG BESTAELLT
0150  007F  82F1  INDEX   AND  HF        MASKA UT SIFFRA 1-5 UR VARUSLAG
0151  0080  9900          STA  INDLAG    OCH ANVAEND SOM INDEX TILL LAGER
            0182
0152  0081  0048          TAX            INDEX TILL X-REG
0153  0082  B500          LDA  ●PLAGER   ANTAL I LAGER TILL A-REG
            01A9
0154  0083  2128          JAZ  SLUTL     OM A=0, HOPPA TILL SLUTL
0155  0084  92FF          SUB  BESANT    SUBTRAHERA BEST ANT
0156  0085  2082          JAM  EJNOG     OM  BEST>LAGER, HOPPA TILL EJNOG
0157  0086  B2FD          LDA  BESANT    BESTAELLT ANTAL TILL A-REG
0158  0087  F20A          JMP  LEVANT    HOPPA TILL LEVANT
0159  0088  0F00  EJNOG   SWM            OM EJ TILLRAECKLIGT I LAGER
0160  0089  F900          JST  *CRLF
            81B3
0161  008A  C6A1          LAP  '!'       TERM-KAR
0162  008B  0F00          SWM
0163  008C  F900          JST  *OTL
            81B1
```

```
PAGE  0004
0164  008D  00F3          DATA TEX5      'BESTAELLD MAENGD...!' UT
0165  008E  0F00          SWM
0166  008F  F900          JST  *CRLF
            81B3
0167  0090  E2F1          LDX  INDLAG    INDEX TILL X-REG
0168  0091  B500          LDA  ●PLAGER   ANTAL I LAGER TILL A-REG
            01A9
0169  0092  9AF0  LEVANT  STA  ANTLEV    ANTAL ATT LEV. TILL MINNET
0170  0093  0F00          SWM            OUTPUT DECIMALTAL
0171  0094  F900          JST  *ODEC     ANTAL ATT HAEMTA UT
            81B0
0172  0095  C6A1          LAP  '!'       TERM-KAR
0173  0096  0F00          SWM
0174  0097  F900          JST  *OTL
            81B1
0175  0098  0104          DATA TEX6      '  ST KAN HAEMTAS VID...!' UT
0176  0099  0F00          SWM
0177  009A  F900          JST  *CRLF
            81B3
0178  009B  E2E6          LDX  INDLAG    INDEX TILL LAGER(A,B,C,D ELLER E)
0179  009C  B500          LDA  ●PLAGER   LAGERINNEHOELL TILL A-REG
            01A9
0180  009D  92E5          SUB  ANTLEV    SUBTRAHERA LEVERERAT ANTAL
0181  009E  9D00          STA  ●PLAGER   REVIDERAT LAGER
            01A9
0182  009F  E2E2          LDX  INDLAG    INDEX TILL PRISLISTAN
0183  00A0  B500          LDA  ●PPRIS    PRISET TILL A-REG
            0175
0184  00A1  0048          TAX            TRANSFER A TO X
0185  00A2  0110          ZAR            NOLLSTAELL A-REG
0186  00A3  1960          MPY  ANTLEV    MULTIPLICERA PRIS OCH ANT BEST
      00A4  0183
0187  00A5  13A8          LRX  1         JUSTERA SVARET
0188  00A6  EAD5          STX  ATTBET    LAGRA UNDAN SVARET
0189  00A7  E2DD          LDX  KUNDN     KUNDNUMMER TILL X-REG SOM INDEX
0190  00A8  B500          LDA  ●PKONTO   TA FRAM KONTOSTAELLNING
            018A
0191  00A9  92D2          SUB  ATTBET    SUBTRAHERA NYSKULD
0192  00AA  9D00          STA  ●PKONTO   KONTOT REVIDERAT
            018A
0193  00AB  F6AA          JMP  KNR       HOPPA TILL 'KUNDNUMMER?' UT
0194  00AC  0F00  SLUTL   SWM            OM SLUT ILAGER
0195  00AD  F900          JST  *CRLF
            81B3
0196  00AE  C6AE          LAP  :AE       TERM-KAR=.
0197  00AF  0F00          SWM
0198  00B0  F900          JST  *OTL
            81B1
0199  00B1  00EC          DATA TEX4      'SLUT I LAGER.' UT
0200  00B2  0F00          SWM
0201  00B3  FBFF          JST  *CRLF
0202  00B4  F6B3          JMP  KNR       HOPPA TILL 'KUNDNUMMER?' UT
0203  00B5  F900  F       JST  LAGST     SUB: GER LAGERSTAELLNING
            01BE
0204  00B6  FAFD          JST  FYLLA     SUB: FYLLER LAGRET
0205  00B7  F6B6          JMP  KNR       HOPPA TILL 'KUNDNUMMER?' UT
0206  00B8  F900  L       JST  LAGST
            01BE
0207  00B9  F6B8          JMP  KNR
0208  00BA  F900  R       JST  RAKN      SUB: GER FAKTURA
            01DD
0209  00BB  F6BA          JMP  KNR       HOPPA TILL 'KUNDNUMMER?' UT
0210                *
0211                *********************************
0212                *
0213                *     DATA DEFINITIONES
0214                *
0215                *********************************
0216  00BC  CBD5  TEX1    TEXT 'KUNDNUMMER (1-30,F,L ELLER R)?'
```

Fig 3.101 b.

```
PAGE  0005
      00BD  CEC4
      00BE  CED5
      00BF  CDCD
      00C0  C5D2
      00C1  A0A8
      00C2  B1AD
      00C3  B3B0
      00C4  ACC6
      00C5  ACCC
      00C6  A0C5
      00C7  CCCC
      00C8  C5D2
      00C9  A0D2
      00CA  A9BF
0217  00CB  C4C9  TEX2   TEXT 'DIN KONTOSTAELLNING AER  KR:'
      00CC  CEA0
      00CD  CBCF
      00CE  CED4
      00CF  CFD3
      00D0  D4C1
      00D1  C5CC
      00D2  CCCE
      00D3  C9CE
      00D4  C7A0
      00D5  C1C5
      00D6  D2A0
      00D7  A0CB
      00D8  D2BA
0218  00D9  C1CE  TEX3   TEXT 'ANTAL OCH VARUSLAG (NX: N=1-9, X=A-E)?'
      00DA  D4C1
      00DB  CCA0
      00DC  CFC3
      00DD  C8A0
      00DE  D6C1
      00DF  D2D5
      00E0  D3CC
      00E1  C1C7
      00E2  A0A8
      00E3  CED8
      00E4  BAA0
      00E5  CEBD
      00E6  B1AD
      00E7  B9AC
      00E8  A0D8
      00E9  BDC1
      00EA  ADC5
      00EB  A9BF
0219  00EC  D3CC  TEX4   TEXT 'SLUT I LAGER.'
      00ED  D5D4
      00EE  A0C9
      00EF  A0CC
      00F0  C1C7
      00F1  C5D2
      00F2  AEA0
0220  00F3  C2C5  TEX5   TEXT 'BESTAELLD MAENGD FINNS EJ I LAGER!'
      00F4  D3D4
      00F5  C1C5
      00F6  CCCC
      00F7  C4A0
      00F8  CDC1
      00F9  C5CE
      00FA  C7C4
      00FB  A0C6
      00FC  C9CE
      00FD  CED3
      00FE  A0C5
      00FF  CAA0
      0100  C9A0
      0101  CCC1
```

```
PAGE  0006
      0102  C7C5
      0103  D2A1
0221  0104  A0D3  TEX6   TEXT ' ST KAN HAEMTAS VID LASTKAJEN!'
      0105  D4A0
      0106  CBC1
      0107  CEA0
      0108  C8C1
      0109  C5CD
      010A  D4C1
      010B  D3A0
      010C  D6C9
      010D  C4A0
      010E  CCC1
      010F  D3D4
      0110  CBC1
      0111  CAC5
      0112  CEA1
0222  0113  CCC1  TEX7   TEXT 'LAGERSTAELLNING:'
      0114  C7C5
      0115  D2D3
      0116  D4C1
      0117  C5CC
      0118  CCCE
      0119  C9CE
      011A  C7BA
0223  011B  C6C1  TEX8   TEXT 'FAKTURA:'
      011C  CBD4
      011D  D5D2
      011E  C1BA
0224  011F  CBD5  TEX9   TEXT 'KUNDNUMMER '
      0120  CEC4
      0121  CED5
      0122  CDCD
      0123  C5D2
      0124  A0A0
0225  0125  A0A0  TEXA   TEXT '    ATT BETALA KR:'
      0126  A0A0
      0127  C1D4
      0128  D4A0
      0129  C2C5
      012A  D4C1
      012B  CCC1
      012C  A0CB
      012D  D2BA
0226  012E  C6C5  TEXE   TEXT 'FELAKTIG BESTAELLNING; FOERSOEK IGEN!'
      012F  CCC1
      0130  CBD4
      0131  C9C7
      0132  A0C2
      0133  C5D3
      0134  D4C1
      0135  C5CC
      0136  CCCE
      0137  C9CE
      0138  C7BB
      0139  A0C6
      013A  CFC5
      013B  D2D3
      013C  CFC5
      013D  CBA0
      013E  C9C7
      013F  C5CE
      0140  A1A0
0227  0141  C6C5  TEXF   TEXT 'FELAKTIGT KUNDNUMMER; '
      0142  CCC1
      0143  CBD4
      0144  C9C7
      0145  D4A0
      0146  CBD5
```

Fig 3.101 c.

```
PAGE  0007
      0147  CEC4
      0148  CED5
      0149  CDCD
      014A  C5D2
      014B  BBA0
0228  014C  B1AD          TEXT '1-30,F,L ELLER R!'
      014D  B3B0
      014E  ACC6
      014F  ACCC
      0150  A0C5
      0151  CCCC
      0152  C5D2
      0153  A0D2
      0154  A1A0
0229  0155  C2D5  TEXG    TEXT 'BUFFER-LAENGD OEVERSKRIDEN; '
      0156  C6C6
      0157  C5D2
      0158  ADCC
      0159  C1C5
      015A  CEC7
      015B  C4A0
      015C  CFC5
      015D  D6C5
      015E  D2D3
      015F  CBD2
      0160  C9C4
      0161  C5CE
      0162  BBA0
0230  0163  D0D2          TEXT 'PROGRAMMET MOESTE LADDAS OM!'
      0164  CFC7
      0165  D2C1
      0166  CDCD
      0167  C5D4
      0168  A0CD
      0169  CFC5
      016A  D3D4
      016B  C5A0
      016C  CCC1
      016D  C4C4
      016E  C1D3
      016F  A0CF
      0170  CDA1
0231  0171  000F  HF      DATA :F        HEX-DEC TALET F
0232  0172  00FF  HFF     DATA :FF       HEX-DEC TALET FF
0233  0173  000A  TIO     DATA 10        DECIMALA TALET 10
0234  0174  00BD  HLIKA   DATA :BD       ASCII-KOD LIKHETS-TKN
0235  0175  0176  PPRIS   DATA $+1       ADRESS-PEKARE TILL PRISLISTA
0236  0176  0005  PRIS    DATA 5
0237  0177  000F          DATA 15
0238  0178  0019          DATA 25
0239  0179  0003          DATA 3
0240  017A  0009          DATA 9
0241  017B        LAGVX   RES  1         TILLF. LAGRING AV VARUSLAG
0242  017C        ATTBET  RES  1         TILLF. LAGRING AV SKULD
0243  017D  02FC  PBUFF   BAC  BUFFIN    ADR.PEK TILL VAENST BYTE I BUFFIN
0244  017E  0000  BUFFIN  RES  4,0       4 ORD RES FOER SVAR
0245  0182        INDLAG  RES  1         INDEX TILL LAGER
0246  0183        ANTLEV  RES  1         ANTAL ATT LEVERERA
0247  0184        BESANT  RES  1         ANTAL BESTAELLDA
0248  0185        KUNDN   RES  1         KUNDNUMMER
0249  0186        ENTAL   RES  1         ENTALSSIFFRAN
0250  0187        CNT     RES  1         COUNTER
0251  0188        COUNT   RES  1         COUNTER
0252  0189        KX      RES  1         KONTONUMMER
0253  018A  018B  PKONTO  DATA $+1       ADRESSPEKARE TILL KONTO
0254  018B  0000  KONTO   RES  30,0      30 ORD RES FOER KONTON
0255  01A9  01AA  PLAGER  DATA $+1       ADRESSPEKARE TILL LAGER
0256  01AA  0000  LAGER   RES  5,0       5 ORD RES FOER LAGER
0257  01AF        VX      RES  1         VARUSLAG
```

Fig 3.101 d.

```
PAGE  0008
0258  01B0        ODEC    REF            SUB: OUTPUT DECIMALTAL
0259  01B1        OTL     REF            OUTPUT LINE TO TTY
0260  01B2        IKL     REF            INPUT LINE FROM KEYBOARD
0261  01B3        CRLF    REF            CARRIGE RETURN/LINE FEED
0262              ***************************************************
0263              *
0264              *  S U B R U T I N E R
0265              *
0266              ***************************************************
0267              *
0268              *
0269              *
0270  01B4  0800  FYLLA   ENT            FYLLER LAGRET
0271  01B5  C705          LAM  5         MINUS 5 TILL A-REG
0272  01B6  9E2F          STA  CNT       LAGRA I COUNTER
0273  01B7  E60E          LDX  PLAGER    ADRESS TILL INDEX-REGISTER
0274  01B8  C61E          LAP  30        +3 TILL A-REG
0275  01B9  9C00  MORE    STA  00        LAGRA 30 AV AKTUELL VARA
0276  01BA  0128          IXR            INKREMENTERA INDEX-REG
0277  01BB  DE34          IMS  CNT       INKR. COUNTER,HOPPA 1 STEG OM 0
0278  01BC  F603          JMP  MORE      MER ATT FYLLA
0279  01BD  F709          RTN  FYLLA     HELA LAGRET FYLLT,RETURN
0280              *
0281              *
0282              *
0283  01BE  0800  LAGST   ENT            GER LAGERSTAELLNING
0284  01BF  C6BA          LAP  ':'       TERM-KAR
0285  01C0  0F00          SWM
0286  01C1  FF10          JST  *OTL
0287  01C2  0113          DATA TEX7      'LAGERSTAELLNING:' UT
0288  01C3  0F00          SWM
0289  01C4  FF11          JST  *CRLF
0290  01C5  C705          LAM  5         MINUS 5 TILL A-REG
0291  01C6  9E3F          STA  CNT       LAGRA I COUNTER
0292  01C7  0110          ZAR            NOLLSTAELL A-REG
0293  01C8  9E46          STA  INDLAG    LAGRA I INDEX-CELL
0294  01C9  C6C1          LAP  :C1       ASCII-KOD FOER A TILL A-REG
0295  01CA  9E1B          STA  VX        LAGRA I VARUSLAG VX
0296  01CB  B61C  LLAG    LDA  VX        VARUSLAG TILL A-REG
0297  01CC  DE1D          IMS  VX        NAESTA VARA I VX
0298  01CD  1357          LLA  8         SKIFTA A-REG 8 STEG VAENSTER
0299  01CE  A65A          IOR  HLIKA     LOGISKT OR MED '='
0300  01CF  9E54          STA  LAGVX     LAGRA I OUTPUT-BUFFER
0301  01D0  C6BD          LAP  '='       TERM-KAR
0302  01D1  0F00          SWM
0303  01D2  FF21          JST  *OTL
0304  01D3  017B          DATA LAGVX     'VX=' UT
0305  01D4  E652          LDX  INDLAG    INDEX TILL X-REG
0306  01D5  B500          LDA  @LAGER    ANTAL I LAGER TILL A-REG
            01AA
0307  01D6  DE54          IMS  INDLAG    INKR. INDEX
0308  01D7  0F00          SWM            SUB: OUTPUT DECIMALTAL
0309  01D8  FF28          JST  *ODEC     LAGERSTAELLNING UT
0310  01D9  FF26          JST  *CRLF
0311  01DA  DE53          IMS  CNT       INKR. COUNTER, HOPPA 1 STEG OM 0
0312  01DB  F610          JMP  LLAG      OM EJV;MR
0313  01DC  F71E          RTN  LAGST     HELA LAGRET VISAT,RETURN
0314              *
0315              *
0316              *
0317  01DD  0800  RAKN    ENT            SKR UT FAKTURA & NOLLST KONTOT
0318  01DE  C71E          LAM  30
0319  01DF  9E58          STA  CNT
0320  01E0  C601          LAP  1         +1 TILL A-REG
0321  01E1  9E58          STA  KX        LAGRA I KONTONUMMER
0322  01E2  E659  NEXT    LDX  KX        INDEX TILL X-REG
0323  01E3  B500          LDA  @PKONTO   KONTOST. TILL A-REG
            018A
0324  01E4  211D          JAZ  NOLL      OM KONTO=0, INGEN FAKTURA
```

```
PAGE  0009
0325  01E5  FA21          JST  RADF      HOPP TILL 15 RAD.FRAMM.
0326  01E6  C6BA          LAP  ':'       TERM-KAR
0327  01E7  0F00          SWM
0328  01E8  FF37          JST  *OTL
0329  01E9  011B          DATA TEX8      'FAKTURA:' UT
0330  01EA  0F00          SWM
0331  01EB  FF38          JST  *CRLF
0332  01EC  FF39          JST  *CRLF
0333  01ED  C6A0          LAP  :A0 TERM-KAR = SPACE
0334  01EE  0F00          SWM
0335  01EF  FF3E          JST  *OTL
0336  01F0  011F          DATA TEX9      'KUNDNUMMER ' UT
0337  01F1  B668          LDA  KX        AKTUELLT KONTONR TILL A-REG
0338  01F2  0F00          SWM            SUB: DECIMALTAL UT
0339  01F3  FF43          JST  *ODEC     KONTONUMRET UT
0340  01F4  C6BA          LAP  ':'       TERM-KAR
0341  01F5  0F00          SWM
0342  01F6  FF45          JST  *OTL
0343  01F7  0125          DATA TEXA      'ATT BETALA KR:' UT
0344  01F8  E66F          LDX  KX        INDEXERING KONTO 1-30
0345  01F9  B500          LDA  @PKONTO   KONTOSTAELLNING TILL A-REG
            018A
0346  01FA  0310          NAR
0347  01FB  0F00          SWM            SUB: DECIMALTAL UT
0348  01FC  FF4C          JST  *ODEC     KONTOSTAELLNING UT
0349  01FD  0F00          SWM
0350  01FE  FF4B          JST  *CRLF
0351  01FF  E676          LDX  KX
0352  0200  0110          ZAR            NOLLSTAELLNING
0353  0201  9D00          STA  @PKONTO   AV KONTOT
            018A
0354  0202  DE79  NOLL    IMS  KX        INKR. INDEX
0355  0203  DE7C          IMS  CNT       INKR. COUNTER FOER ANTAL KONTON
0356  0204  F622          JMP  NEXT      OM EJ ALLA KONTON FOETT RAEKNING
0357  0205  FA01          JST  RADF      HOPPA TILL 15 RAD.FRAMM.
0358  0206  F729          RTN  RAKN      OM KLAR,RETURN
0359              *
0360              *
0361              *
0362  0207  0800  RADF    ENT            15 RADERS FRAMMATNING
0363  0208  C70F          LAM  15        MINUS 15 TILL A-REG
0364  0209  9E81          STA  COUNT     LAGRA I COUNTER
0365  020A  0F00          SWM
0366  020B  FF58          JST  *CRLF
0367  020C  DE84          IMS  COUNT     INKR. COUNTER
0368  020D  F603          JMP  $-3       OM EJ KLAR
0369  020E  F707          RTN  RADF      OM KLAR, RETURN
0370              *
0371              *
0372              *
0373  020F  0800  CHBFRL  ENT            KOLL OM BUF-LAENGD OEVERSKRIDEN
0374  0210  C708          LAM  8         8 BYTES RESERVERADE FOER INPUT
0375  0211  9E8A          STA  CNT       MINUS 8 TILL COUNTER
0376  0212  E695          LDX  PBUFF     BYTE-ADRESS TILL X-REG
0377  0213  0E00  LOP1    SBM            SET BYTE MODE
0378  0214  B400          LDAB @0        BYTE TILL A-REG
0379  0215  0F00          SWM            SET WORD MODE
0380  0216  C08D          CAI  :8D       BYTE=CARRIGE RETURN?
0381  0217  F708          RTN  CHBFRL    JA,RETURN
0382  0218  0128          IXR            NEJ, INKR X-REG
0383  0219  DE92          IMS  CNT       INKR. COUNTER
0384  021A  F607          JMP  LOP1      OM EJ ALLA 8 KAR. KOLLADE
0385  021B  0F00  ERRG    SWM            INGET CR I INPUT BUFFER
0386  021C  FF69          JST  *CRLF
0387  021D  C6A1          LAP  '!'       TERM-KAR
0388  021E  0F00          SWM
0389  021F  FF6E          JST  *OTL
0390  0220  0155          DATA TEXG      'BUFFERLAENGD OEVER...!' UT
0391  0221  0F00          SWM
```

```
PAGE  0010
0392  0222  FF6F          JST  *CRLF
0393  0223  0801          STOP 1
0394                      END
0000 ERRORS
```

```
PAGE  0011
    ANTLEV  0183    ATTBET  017C    BESANT  0184    BESTIN  004F
    BUFFIN  017E    CHBEST  0069    CHBFRL  020F    CHKNR   0029
    CH1     0040    CNT     0187    COUNT   0188    CRLF    01B3
    EJNOG   0088    ENTAL   0186    ERRE    0060    ERRF    0020
    ERRG    021B    FYLLA   01B4    F       00B5    HFF     0172
    HF      0171    HLIKA   0174    IKL     01B2    INDEX   007F
    INDLAG  0182    KNR     0001    KONS    0047    KONTO   018B
    KUNDN   0185    KX      0189    LAGER   01AA    LAGST   01BE
    LAGVX   017B    LEVANT  0092    LLAG    01CB    LOP1    0213
    L       00B8    MORE    01B9    NEXT    01E2    NOLL    0202
    ODEC    01B0    OTL     01B1    PBUFF   017D    PKONTO  018A
    PLAGER  01A9    PPRIS   0175    PRIS    0176    RADF    0207
    RAKN    01DD    R       00BA    SLUTL   00AC    TEXA    0125
    TEXE    012E    TEXF    0141    TEXG    0155    TEX1    00BC
    TEX2    00CB    TEX3    00D9    TEX4    00EC    TEX5    00F3
    TEX6    0104    TEX7    0113    TEX8    011B    TEX9    011F
    TIO     0173    VX      01AF
```

Fig 3.101 Kassaprogrammet efter assembleringen.

AAI	addera till A direkt (immediate)
AND	logiskt AND med A
CAI	jämför med A direkt (hoppa om olika)
IMS	inkrementera M (hoppa över om noll)
IOR	logiskt OR med A
IXR	inkrementera X
JAG	hoppa om A > 0
JAM	hoppa om A < 0
JAL	hoppa om A ≤ 0
JAZ	hoppa om A = 0
JMP	hoppa
JST	hoppa och lagra (till subrutin)
LAM	ladda A direkt (negativt)
LAP	ladda A direkt (positivt)
LDA	ladda A
LDAB	ladda A med BYTE
LDX	ladda X
LLA	(logiskt) vänsterskift för A
LRA	(logiskt) högerskift för A
LRX	(logiskt) högerskift för X
MPY	multiplicera X med minne och addera A.
NAR	byt tecken på A
SAI	subtrahera från A direkt
SBM	sätt BYTE MODE (dvs 8 bitar)
STA	lagra A
STOP	stoppa (med operand)
STX	lagra X
SUB	subtrahera från A
SWM	sätt WORD MODE
TAX	överför från A till X
TXA	överför från X till A
ZAR	nollställ A

Fig 3.102 Instruktionerna i kassaprogrammet.

BAC	adresspekare för BYTE
DATA	placera efterföljande data i minnet
END	slut på assemblerprogrammet
ENT	början på subrutin (enter)
RES	reservera ett antal celler i minnet
RTN	slut på subrutin (return), återvänd till huvudprogrammet
TEXT	placera bokstäver (textsträngar) i ASCII-kod i minnet
REF	tillfällig lagring av adresspekare

Fig 3.103 Direktiven i kassaprogrammet

3.5.4 NÅGRA DETALJER I ASSEMBLERPROGRAMMET

Låt oss följa datorns operationer när den exekverar början av kassaprogrammet, dvs från rad 24 och framåt.

Rad 24

Första instruktionen är JST, hoppa och lagra. Här sker ett hopp till subrutinen FYLLA. Adressen anges här med etiketten FYLLA och enligt sammanställningen på sid 11 i programmet ligger FYLLA på programsteg Ø1B4 (som vi hittar på rad 270). Assemblern har välvilligt lagt in denna adress mellan raderna 24 och 25 för att förenkla arbetet både för oss och för sig själv.

I flödesplanen (fig 3.99) har vi satt radnummer för subrutiner inom parentes. Subrutinen FYLLA fyller lagrets dvs cellerna Ø1AA, Ø1AB, Ø1AC, Ø1AD och Ø1AE i minnet (rad 256) med vardera siffran 30. Vi får alltså 30 st av varje varuslag (A-E) lagrade i minnet. Vi ska strax återkomma till detaljerna i denna subrutin, men låt oss först arbeta vidare en stund i huvudprogrammet.

Rad 25-30

Datorn ska nu börja sin affärsverksamhet med att skriva "KUNDNUMMER (1-30, F, L ELLER R)?" och därefter skall datorn avvakta den första kundens ankomst till terminalen. Parantesen har tillkommit för att informera kunderna (och affärsföreståndaren) om vilka svar som kan accepteras.

Innan datorn påbörjar utskriften förvissar den sig om att texten skall börja längst till vänster på ny rad. Det är en ofta återkommande operation och finns därför som ett subrutin i ALPHA LSI:s software support. Rutinen heter CRLF (carriage return, line feed) och kräver att datorn arbetar i WORD MODE. Subrutinen föregås därför (för säkerhets skull) av instruktionen SWM (set word mode).

Genom att ge kommandot

JST *CRLF

får vi ett hopp till subrutinen. Asterisken anger att detta sker via indirekt adressering (Ø1B3 på rad 261).

När subrutinen CRLF är exekverad skall alltså texten skrivas ut på TTY:n. Texten finns lagrad som TEX1 på rad 216. Inklusive mellanslaget blir det allt som allt 30 tecken ("karaktärer") som alltså kräver 15 celler i minnet.

Textsträngen börjar med :CBD5, dvs KU, och slutar med :A9BF, dvs)?. ASCII-koden återges i fig 3.78c.

Sista tecknet i en textsträng kallas ofta "sluttecken" eller "termineringskaraktär". Det har som vi nu ska se en viss betydelse vid utskriften.

Datorn börjar utskriftsoperationen med att sätta slutteckenet :BF (dvs frågetecknet) i ackumulatorn med instruktionen

LAP :BF (ladda A direkt med :BF).

Med instruktionen SWM sätts datorn liksom tidigare i WORD MODE (SWM förekommer på många ställen i programmet och vi kommenterar den inte i fortsättningen).

Därefter hoppar datorn till subrutinen OTL (skriv rad på TTY:n).

Omedelbart efter hoppinstruktionen (JST * OTL) ges adressen till textsträngen med direktivet DATA TEX1. Datorn skriver nu i tur och ordning ut alla tecken ur minnet och fortsätter tills den hittar slutteckenet (:BF). Då avbryter datorn utskriften och därmed är textsträngens utskrift fullbordad.

Rad 31-36

Förfrågan om kundnummer är nu utskriven på TTY:n och vi befinner oss på rad 31 i programmet. Vi känner där igen sekvensen SWM, JST * CRLF som ger oss ny rad och återgång till radens början.

Nu skall datorn invänta ett svar och det gör den med hjälp av subrutinen IKL (input line from keyboard). Frågan är nu hur datorn ska veta när kunden är färdig med sitt svar.

I programrubriken (rad 21) står TERMINERINGS-KAR: CARRIAGE RETURN. Därmed avses att alla meddelanden vi ger datorn skall avslutas med tryck på RETURN-knappen.

När IKL-rutinen exekveras skall termineringskaraktären finnas tillgänglig i A-registret. Därför föregås subrutinen IKL av LAP :8D där :8D just är ASCII-koden för RETURN.

Omedelbart efter JST * IKL (dvs på rad 36) anges var i minnet som svaret skall placeras. Svaret skall tydligen placeras på adressen BUFFIN som vi finner på rad 244 där 4 celler är reserverade för detta ändamål. (Plats har alltså reserverats för ett svar som innehåller maximalt 8 tecken.)

Rad 37-43

På rad 37 följer en subrutin som kontrollerar om svarets längd överstiger den reserverade platsen i minnet. Om så är fallet kan programmet bli förstört varför felmeddelande ges.

Vi går vidare till rad 40. Här laddas svaret i A (LDA BUFFIN) och nu gäller det att identifiera vad kunden har meddelat datorn.

Genom att skifta svaret 8 steg till höger (rad 41) får vi den vänstra BYTEN i svaret placerad i ackumulatorns positioner 0-7. Vi jämför nu innehållet i dessa positioner med ASII-koden för F, L och R.

CAI betyder "jämför A direkt med följande". I detta fall är "följande" bokstaven F, men den måste anges i ASCII-kod. Assemblern läser 'F' som ASCII-koden för F (dvs :C6), men instruktionen hade lika bra kunnat skrivas CAI :C6 (liksom LAP :BF på rad 27).

Om jämförelsen vid CAI-instruktionen ger resultat (dvs i vårt fall om innehållet i A är lika med CAIinstruktionens 'F') exekveras efterföljande instruktion (rad 43) som är ett

hopp till adressen F (JMP F). Adressen F hittar vi på rad 203 och här exekveras två subrutiner efter varandra.

Rad 44-52

Om jämförelsen ej ger resultat - och vi antar nu att detta är fallet - hoppar programräknaren ett steg (till rad 44). Där följer nu en jämförelse med L.

Hamnar vi på rad 48 har svaret tydligen varken varit F, L eller R. Det måste då tydligen vara ett kundnummer (om det överhuvudtaget är acceptabelt!).

Svaret som finns lagrat på adress BUFFIN laddas åter in till ackumulatorn (LDA BUFFIN på rad 48). Låt oss anta att svaret är

3 RETURN
(dvs kundnummer 3).

För att få reda på om det är en eller två siffror i svaret börjar datorn med att "maska ut" (dvs enbart ta med) högra BYTEN (LSB) i svaret. Den bör ju vara RETURN om svaret är ensiffrigt.

Genom att göra en AND-funktion med :FF (dvs med talet 0000000011111111) och acumulatorn A kommer högra BYTEN i A att bli utmaskad. (Se fig 3.23).

En jämförelse med instruktionen (rad 50)

CAI :8D

visar om högra BYTEN är RETURN (:8D). Om så är fallet exekveras hoppinstruktionen

JMP CH1

på rad 51 och vi hamnar på rad 87 i programmet.

Rad 87-93

På rad 87 laddas svaret åter till A med LDA BUFFIN och därefter vidtas en kontroll om svaret håller sig inom givna gränser, dvs 1-9.

Om svaret i A är större än 9 ger en subtraktion med 9 (SAI :B9) positiv differens. Med JAG, dvs hoppa om A > 0, sker härvid ett hopp till adressen ERRF som ger felutskrift.

Instruktionen AAI 9 adderar 9 direkt. Eftersom vi på rad 89 subtraherat med :B9 och på rad 91 adderat med :9 har vi fått talet i första BYTEN omvandlat till en siffra i binär kod (i stället för den ASCII-kod som lagrats med LDA BUFFIN på rad 87).

Vi antog tidigare att de aktuella kundnumret var 3 och det har alltså klarat testerna på rad 87-92. Det kontrollerade kundnumret lagras nu (i binär kod) på adress KUNDN (rad 93).

Rad 94-97

Nu är allt klart för utskrift av kontoställningen på konto nr 3. Först skrivs en textsträng (TEX2) ut på känt sätt (rad 94-97). Därefter laddas X-registret med det aktuella kundnumret (LDX KUNDN) som alltså är 3.

Nu skall A-registret laddas med innehållet i kundkontot nr 3. Här kan vi nu använda indexerad adressering (Se avsnitt 3. 2. 6)

Med instruktionen

LDA @PKONTO

laddas A med innehållet i cellen med adressen "PKONTO+X", dvs :Ø18A + :3 = :Ø18D. Detta är den tredje av de reserverade cellerna. Tecknet @ anger att vi här använder indexerad adressering.

När vi nu fått innehållet från kundkonto nr 3 till A-registret är allt klart för utskrift. Men vi måste nu ta hänsyn till att datorn lagrar tal binärt medan utskriften skall ske decimalt. Vi använder därför subrutinen (rad 101)

ODEC, output decimal (skriver ut 5 decimala siffror samt tecken).

Vi har nu följt några delar av huvudprogrammet. Vi överlämnar resten av programmet för läsarens egen "forskningsresa" med hjälp av kommentarerna i programmet och programmeringsmanualen till ALPHA LSI.

En subrutin

Innan vi lämnar assemblerprogrammet ska vi kort kommentera en av subrutinerna. Ett program blir ofta både enklare att göra och lättare att förstå om olika deluppgifter läggs som subrutiner. FYLLA är en sådan subrutin och den återfinns på raderna 270-279.

Varje subrutin börjar med direktivet ENT (dvs enter). Här lagras återhoppsadressen till huvudprogrammet (JST = hopp och lagra. Återhoppsadress = adressen till cellen omedelbart efter JST-instruktionen).

Varje subrutin måste avslutas med direktivet RTN (dvs return) följt av adressen till den cell där återhoppsadressen är lagrad. RTN FYLLA på rad 279 betyder alltså att återhoppet sker till den adress som är lagrad i cell FYLLA (på rad 270).

FYLLA-rutinen laddar de fem lager-cellerna med 30. Detta sker med indexerad adressering. Genom att lagra talet -5 i en "counter" (CNT) på rad 272 och sedan inkrementera denna counter (IMS CNT) för varje påfyllning (LAP 30) laddar man samtliga 5 celler med 30. När samtliga är fyllda är countern 0 och då hoppar IMS-instruktionen över rad 278 (JMP MORE) och till RTN på rad 279. Subrutinen är därmed utförd.

3.5.5 FRÅN KÄLLKOD TILL KASSAPROGRAM

Den som nu vill bita i en svårare uppgift kan skriva ner källkoden till kassaprogrammet, dvs kolumn 4-6 i fig 3.101, och själv köra assembleringen i OMEGA. Programmet kräver rätt stor plats vid assembleringen och det är därför lämpligt att utvidga minnet till 16 kord. Denna minnesarea disponeras så att LAMBDA BUFFER och följande områden i fig 3.93 flyttas till :39ØØ och därmed blir USER AREA avsevärt utökad.

När assembleringen utförts laddas programmet tillsammans med de yttre subrutinerna från TUP med hjälp av LAMBDA. Slutligen kan en binärremsa tas ut med hjälp av BDP/VER.

3.5.6 KASSAPROGRAMMET I BASIC

Eftersom vi nu ökat vår minnesarea till 16 kord kan vi använda ett mer utrymmeskrävande programspråk. BASIC är ett språk som är lätt att lära och som därför används flitigt.

Vi kan nu ladda in BASIC4, ett i Uppsala utvecklat program för ALPHA LSI. BASIC-tolken kan beställas från Scandia Metric och levereras i form av en hålremsa som enkelt laddas in med AUTOLOAD.

Den första bekantskapen med BASIC kan lämpligen göras med någon av de många BASIC-böcker som förekommer på bokmarknaden. En bra bok är:

Vi lär ADB och BASIC
Studentlitteratur 1973.

Kassaprogrammet kan skrivas i BASIC och ett förslag framgår av fig 3.104. Pröva själv!

Lägg märke till att kassaprogrammet blir kortare och enklare när det skrivs i BASIC. Det kräver emellertid större minnesutrymme. BASIC-programmet blir lätt förstört eftersom operatören (dvs kunden) här har access till själva programmet. Assembler-programmet är däremot mer svåråtkomligt för kunden - speciellt om skyddsrutiner inkluderas i programmet.

```
00010 REM   KASSAPROGRAMMET
00020 REM   FROEN KASSATERMINALEN KAN BESTAELLAS
00030 REM   ANTAL:              1-9
00040 REM   VARUSLAG:           A,B,C,D ELLER E
00050 REM   OM KUNDNUMMER=0:
00060 REM    L  GER LAGERSTAELLNING
00070 REM    F  GER LAGERSTAELLNING
00080 REM       OCH FYLLER LAGRET
00090 REM    R  SKRIVER UT FAKTURA SAMT
00100 REM       NOLLSTAELLER KONTON
00110 REM   PRIS: VARA          A=  5 KR
00120 REM                       B= 15 KR
00130 REM                       C= 25 KR
00140 REM                       D=  3 KR
00150 REM                       E=  9 KR
00160 REM    ****************************************
00170 DIM K(31),L(6),P(6),X$(6),V$(1),S$(1)
00180 LET X$(1;1)="A"
00190 LET X$(2;2)="B"
00200 LET X$(3;3)="C"
00210 LET X$(4;4)="D"
00220 LET X$(5;5)="E"
00230 LET P(1)=5
00240 LET P(2)=15
00250 LET P(3)=25
00260 LET P(4)=3
00270 LET P(5)=9
00280 GOSUB 1200
00290 GOSUB 1250
00300 PRINT "KUNDNUMMER (1-30;";
00310 PRINT " SVAR=0 GER NY FROEGA)?"
00320 INPUT K1
00330 REM   OM L,F ELLER R OENSKAS
00340 IF K1=0 THEN 780
00350 FOR I=1 TO 30
00360   REM   OM GODKAENT KUNDNUMMER
00370   IF K1=I THEN 420
00380 NEXT I
00390 REM   OM FELAKTIGT KUNDNUMMER
00400 GOTO 1090
00410 REM   KUNDNUMMER 1-30
00420 PRINT "DIN KONTOSTAELLNING AER:  ";
00430 PRINT K(K1)
00440 PRINT "ANTAL OCH VARUSLAG";
00450 PRINT "(N,""X"";N=1-9,""X""=""A""-""E"")?"
00460 INPUT A1,V$
00470 FOR I=1 TO 9
00480   IF A1=I THEN 520
00490 NEXT I
00500 GOTO 1110
00510 REM   ANTAL BESTAELLDA=1-9
00520 IF V$(1;1)="A" THEN 590
00530 IF V$(1;1)="B" THEN 610
00540 IF V$(1;1)="C" THEN 630
00550 IF V$(1;1)="D" THEN 650
00560 IF V$(1;1)="E" THEN 670
00570 GOTO 1110
00580 REM   J=INDEX TILL LAGER & PRISLISTA
00590 LET J=1
00600 GOTO 680
00610 LET J=2
00620 GOTO 680
00630 LET J=3
00640 GOTO 680
00650 LET J=4
00660 GOTO 680
00670 LET J=5
00680 IF L(J)=0 THEN 1130
00690 IF A1>L(J) THEN 1160
```

```
00700 LET L1=A1
00710 PRINT L1;
00720 PRINT "  ST KAN HAEMTAS VID LASTKAJEN!"
00730 LET L(J)=L(J)-L1
00740 LET P1=L1*P(J)
00750 LET K(K1)=K(K1)-P1
00760 GOTO 300
00770 REM   L,F ELLER R OENSKAS
00780 PRINT "INPUT ''F'',''L'' ELLER ''R'''"
00790 INPUT S$
00800 IF S$(1;1)="L" THEN 850
00810 IF S$(1;1)="R" THEN 870
00820 IF S$(1:1)="F" THEN 1040
00830 GOTO 1070
00840 REM   OM LAGERSTAELLNING OENSKAS
00850 GOSUB 1300
00860 GOTO 300
00870 PRINT
00880 PRINT
00890 FOR I=1 TO 30
00900   IF K(I)=0 THEN 1020
00910   PRINT "FAKTURA:"
00920   PRINT
00930   PRINT "KUNDNUMMER ";
00940   PRINT I;
00950   PRINT "   ATT BETALA KR:   ";
00960   LET C=-K(I)
00970   PRINT C
00980   LET K(I)=0
00990   FOR J=1 TO 15
01000     PRINT
01010   NEXT J
01020 NEXT I
01030 GOTO 300
01040 GOSUB 1300
01050 GOSUB 1200
01060 GOTO 300
01070 PRINT "FELAKTIGT SVAR; L,R ELLER F GILTIGT!"
01080 GOTO 300
01090 PRINT "FELAKTIGT KUNDNUMMER;1-30 ELLER 0 GILTIGT!"
01100 GOTO 300
01110 PRINT "FELAKTIG BESTAELLNING;FOERSOEK IGEN!"
01120 GOTO 440
01130 PRINT "SLUT I LAGER."
01140 GOTO 300
01150 REM   BESTAELLNING>LAGER
01160 PRINT "BESTAELLD MAENGD FINNS EJ I LAGER!"
01170 LET L1=L(J)
01180 GOTO 710
01190 REM   SUBRUTIN FYLLA
01200 FOR I=1 TO 5
01210   LET L(I)=30
01220 NEXT I
01230 RETURN
01240 REM   SUBRUTIN RESKON
01250 FOR I=1 TO 30
01260   LET K(I)=0
01270 NEXT I
01280 RETURN
01290 REM   SUBRUTIN LAGST
01300 PRINT "LAGERSTAELLNING:"
01310 FOR I=1 TO 5
01320   PRINT X$(I:I);
01330   PRINT "=";
01340   PRINT L(I)
01350 NEXT I
01360 RETURN
01370 END
TYPE
```

Fig 3. 104 Kassaprogrammet skrivet i BASIC.

4. Vidga horisonten

Vi har nu vandrat igenom ett omfattande ämnesområde med minidatorn ALPHA LSI som ledtråd. Vandringen har fört oss från enkla kretsar som kombinerats och integrerats och blivit komplexa elektroniska system. Vi har följt operationerna i datorn från mikronivån - dvs hanteringen av data i datorns inre trafiksystem - till assemblerspråkets kryptiska och strama kommandon.

Givetvis är det mycket som inte blivit förklarat eller genomlyst på denna vandring. Men har vår vandring lämnat ett spår över fältet kan vi nu trampa upp det spåret till en stig, för nu vet vi vart det leder.

Kanske är det hårdvaran och elektroniken som intresserar. Om bakgrunden här saknas kan nedanstående ref. 1 och 2 ge en lämplig introduktion. För den som vill gå vidare kan ref 3 och 4 ge merläsning av lagom svårighetsgrad.

Kanske är det mjukvaran som intresserar. Då finns det en rad böcker på Studentlitteratur och en av dem har vi redan rekommenderat, ref 5.

Kanske är det datorns inre arkitektur som står i centrum för vårt intresse. Per-Erik Danielssons bok "Datateknik, Arkitektur" från Linköpings högskola (ref 6) är då lämplig merläsning.

Det finns givetvis en mängd amerikanska böcker om minidatorer och det kommer ständigt fler. Ref 7 och 8 ger två exempel.

Litteraturreferenser

1. Markesjö, Elektronik, Inledande kurs A. Esselte Studium 1972
2. Markesjö, Elektronik, Inledande kurs B. Esselte Studium 1972
3. Markesjö, Pulskretsar, Esselte Studium, 1976
4. Markesjö, Digitala kretsar, Esselte Studium, 1976
5. Siro (red), Vi lär oss ADB och BASIC, Studentlitteratur 1973
6. Danielsson, Datateknik, Arkitektur, Linköpings högskola, Institutionen för Systemteknik, 58183 Linköping
7. The Value of Power, General Automation, Anaheim, Calif, 1974.
8. Korn, Minicomputers for engineers and scientists, Mc Graw Hill, New York 1973.

OMEGA COMMAND SUMMARY

CONTROL

I.	Initialize OMEGA -- clear Buffer and reset last Input Line read to Ø.
B.	Buffer clear.
Ehhhh.	End of memory set to hexadecimal address.
M.	Memory available displayed in decimal words.
X.	Execute assembler.
XE.	Error list only.
XL.	Library Format for Object Program -- no EOF.
X2.	Pass 2 again.

BUFFER EDITING

AF.	Add after last line.
Am.	Add after Line m.
Dm.	Delete Line m.
Dm n.	Delete Lines m thru n.
Lm.	List Line m.
Lm n.	List Lines m thru n.
P m.	Punch Line m.
P m n.	Punch Lines m thru n.
PL m n.	With leader
PT m n.	With trailer.
PLT m n.	With leader and trailer.

INPUT EDITING

Rm.	Read thru Line m, and add to end of Buffer.
Sm.	Skip Line m, after reading thru Line m-1.
Sm n.	Skip Lines m thru n, after reading thru Line m-1.
Tm.	Reset last Input Line read to m.

LOGICAL DEVICES

Cd. Connect devices:

Source Input	Punch Output	List Output
I1 Teletype Keyboard	O1 Teletype	L1 Teletype
I2 Teletype Paper Tape	IØ EOF Now	L2 Data Products
I3 HS Paper Tape	O3 HS Paper Tape	L3 Centronics
I4 Card Reader	O4 HS Paper Tape (DIO)	L4 Data Products (DIO)
I5 Buffer Lines to X.	OØ No Punching	L5 Centronics (DIO)
I6 Card Reader (DIO)		LØ No Listing
I7 HS Paper Tape (DIO)		

DEBUG COMMAND SUMMARY

A.	Display pseudo A register.
Av.	Set pseudo A register to value v.
Ba.	Continue breakpoint to location a.
Ba,b.	Continue breakpoint to location (a or b).
Ba b.	Breakpoint from location a to b.
Ba b,c.	Breakpoint from location a to location (b or c).
Ca b c.	Copy locations a through b at c and following.
Fa b v.	Fill locations a through b with value v.
Ia.	Inspect location a.
Ja.	Jump to location a.
Ma.	Modify memory starting at location a.
O.	Display pseudo O register.
Ov.	Set pseudo O register to value v.
Pa b.	Print locations a through b.
Rn.	Display relocation register Rn.
Rn v.	Set relocation register Rn to value v.
Sa b v.	Search locations a through b for value v.
Sa b v m.	Search for value v using mask word m.
T.	Enable console interrupt (TRAP).
Tn.	Enable console interrupt and enable interrupts.
X.	Display pseudo X register.
Xv.	Set pseudo X register to value v.

Autoload

RESET
To relocate, set X = load address
Enter sum of options in Sense Register:

Mode \ Device	TTY	HSPT	MT	Cassette	Disk
Load ABS	:Ø	:1	:2	:3	:4
Load Rel	:8	:9	:A	:B	:C

For Load and Go, set SENSE Switch
Ready Device
AUTO

Binary Dump/Verify (BDP/VER)

Load BDP; set P = BDP rel. zero
Set A = Initial location
Set X = Last location
Enter sum of options in Sense register:

Mode \ Device	TTY	HSPT
Punch Abs	:Ø	:1
Punch Rel	:8	:9

To verify, add :4
To suppress EOF, add :2
For transfer address, set SENSE switch
RUN
If Halt (I = :Ø8Ø2), set A = transfer address, RUN

Object Loader (LAMBDA)

Load LAMBDA; set P = LAMBDA zero
Set A = Relocation Bias or zero
Set X = Base Page Bias or zero
Enter sum of options in Sense register:

Load Mode \ Print Symbols	Defined and Undefined	Defined Only	Undefined Only	Neither
Library	:Ø	:2	:4	:6
Unconditional	:8	:A	:C	:E

Ready tape in reader (TTY or HSPT)
RUN

Sakregister
Svenskt

A-buss 92,95,101
A-registret 56,61
absolut 152
access 32
ackumulator 56
addend 24,29
addera 23
adderaren 24,72
addition 23,31
adress 30,78
adressavkodare 40,84
adressbuss 95
adressering 118
adresseringsmod 119
adressingången 38
adresspekare 118
ALU 27, 72, 76
AND-funktionen 76
anpassningskretsar 43
arbetsmod 68
aritmetisk logisk enhet 72
aritmetiska operationer 23
arkitektur 97,100
ASCII 133
ASR 33,126
assembler 151
assemblering 161
assemblerprogram 159
assemblerprogrammering 151
augend 20,24
AUTO ECHO 147
AUTOLOAD 140,152
avbrott 69
avkodare 17,27
avläsning 61
avspelning 44

B-bussen 92
bandminnet 42
BASIC 167
BASIC-tolkar 168
baud 133
BCD-kod 21
BDP/VER 151
binär siffra 16
binärkod 163
binärremsa 163
bipolära transistorer 19
bitledning 85
block 43
blockschema 21,100
brytpunkter 167
buffer 138,156
buffertregister 138
bussar 90
byte 38
C-buss 101
C-tangent 24
carriage return 144,146
carry 73
celler 83
CMOS 19
codebars 128
conditional output 141
connect-kommandon 157
counter 187
CPU 52
CPU-kort 54
CR 144
current loop 133

D-buss 95,101
dataregister 60
databuss 95
dataingång 38
datamaskin 10
dataminne 38
datautgången 40
dataväljare 109
dator 10
datorsystem 11
DEBUG 151
differens 25
digitala system 16
direktinstruktion 145
direktiv 160
disk 168
display 22
distributor 128
driver/receiver 97
driver 97
drivsteg 97
dumpa 163
dynamiska minnen 88

eko 147
elektrolytkondensator 124
enable console 58
etikett 163
ett 16
ett-komplement 26
exekvera 69,109
exponent 38

FA 73
fack 83
fast logik 41,103
felsökning 167
ferritkärna 57
fil 43
fjärrskrivmaskin 126
flödesplan 173
formatering 43
FORTRAN 167
funktion 174
förflytta data 89

grind 16

HA 73
halvadderare 73
halvledarminne 57
handskakning 136
hang 123
hexadecimal 60
hexadecimal kod 60
hopp 35
hopp och lagra 121
hoppinstruktion 121
hoppvillkoren 121
huvudprogram 35
huvudströmställaren 58
hämta 109
högnivåspråk 167
hölje med fläkt 54
hållkretsar 80

I/O-bussen 95
I/O-enhet 43,126
I/O-instruktion 141
IC 19
IC-kretsar 18
IEC 10
immediate 145
in 174
in-enheter 23
in-ut-enhet 43,126
indexerad adressering 118
indikator 23
indirekt adressering 118
inhopp 174
initiera TTY:n 142
inskrivning 61
inspelning 44
instruktioner 160
instruktionsadress 64
instruktionskod 104
instruktionslista 35,62
instruktionstangenter 20
integrated circuit 19
integrerade 13
interface 43,136
interrupt 69

jump dollar 148

kalkylatorns blockschema 30
kapacitet 44
kassaprogrammet 171
kassettbandstation 42
keramikkondensator 124
kiloord 57
klart för start 69
klocka 72
kodstänger 128
kombinationskrets 17
kontrollbuss 95
kontrollingång 17,40,90
kontrollpanel 54,58
kontrollsignaler 133
konversationsassembler 151
kopplingsbox 136
kraftförsörjningen 122
krets 13
kringutrustning 43
kristall 38
kylning 122
källkod 161,163
källprogram 161
kärnminne 56
köra 69

label 163
ladda 165
lagra 30,78
LAMBDA 151
LAMBDA buffer 154
latch 78,80
LF 144
line feed 144,146
lista 161
logiska funktioner 77
LSB 73

LSI-krets 13
läsminne 38,83

M-registret 56,61
mantissa 38
mark 133
mask 77
maska 76
maskininstruktion 65,104,116
maskinkod 65
maskinvara 150
matningsspänning 124
matris 86
maxibuss 95
memosymbol 62,65
mikrodator 11
mikroinstruktion 24
mikroprocessor 11
mikroprogram 104
mikroprogramminne 83
minidator 11
minne 30,56
minneskort 54
minnesmatris 85
minnesreferande 62
minnesregister 31
minnessiffra 73
minst signifikanta biten 26
minuend 25
mjukvaruregister 156
moderkort 54,55
mottagare 97
MS-vippa 75
multiplex 95,108
MUX 108
mästare 75

NAKED MINI 53
noll 16
nollställningstangent 20
nätdel 54
nätaggregat 122

objektkod 163
objektremsa 163
OEM 54
oktader 38
oktal kod 34
oktetter 38
OMEGA 151
omkopplare 128
operandadress 62,116,118
operand 62
operationskod 102,116,118
operationsorder 62
operativsystem 168
ord 38
ordledning 40,85
ordlängd 21
OS 168

P-registret 56 61
paritetsbiten 133
pekare 118
power fail 123
primärminne 57
processor 52
program 27,63
programmerbar 32
programmerbar kalkylator 29
programmering 86
programminne 32,38
programräknare 34,56
programstyrd 45
programvara 150
pull-up-motstånd 94

R-bussen 90
radframmatning 144
RAM 38,87
ready console 58
realtidsklocka 140
register 21,56,78,82,157,158
registerfil 88
relativ adressering 118
relokera 149,152
remsläsare 126
remsstans 126
resultatbussen 90
ROM 38,83
ROS 83
RTC 140
RUN 68
rutiner 27
räknare 10
räknedosan 13
räkneenhet 72
räknemaskin 10

scratchpad 118
scratchpadkopia 154
SEK 10
sekundärminne 57
select 141
SEN 10
sense 141
SENSE-register 60

siffertangenter 20
signalingång 17
sjusegmentsmönster 22
självadresserande programverk 107
skiftregister 22
skivminne 42,168
skriv-läsminne 38,87
skrivkommando 142
slav 75
slinga 68,148
sluttecken 184
software support 150
space 133
start 163,174
starta 69
startbit 133
startknapp 68
statiska halvledarminnen 88
stega 69
stop-order 33
stopp 174
stoppa 69
stoppbit 133
stopptecken 163
strömslinga 133
strömspikar 124
styrenhet 99,102
styrorder 111
subrutin 35,174,187
subtrahend 25
subtraktion 25
summa 24
sändare 97

tangentbord 126
teckenskrivare 126
termineringskaraktär 171,184
testa 47
textsträng 177,184
tristate 94
trumminne 42
tryckverk 128
TTL 19
TTY-interfacet 137
TTY-snitt 137
TTY:n 126
TUP 177
två-komplement 26
typhuvudet 128
typkoder 163

UART 138
UNIBUS 95
user area 154

ut 174
ut-enhet 23
uthopp 174

vagnretur 144
vinkel 157
vippa 16
volatil 42

öppen kollektor 92
överföringssiffra 73

Sakregister
Engelskt

access 32
acknowledge 111
address pointer 118
architechure 27
aritmetic logic unit 27
ASCII 133
AUTO ECHO 147

branch 35
breakpoints 167
bus 90

calculator 10
carriage return 144
carry 73
CE 24
central processing unit 52
chip 38
clear 20
clear entry 24
code 27
codebars 128
computer 10
conditional output 141
conditional word input instruction 147
conditional word output instruction 147
console data register 60
console sense register 60
control unit 27,72
counter 187
CPU 37,52
CR 144
current loop 133
data 27
decoder 27
display 22
distributor 128,133
driver/receiver 97
drivers 97
endless tape 45
execute 109

FA 73
fetch 109
flip flop 16
full adder 73

gate 16

HA 73
halfadder 73
handshake 136
hang 123

I/O 43
IC 19
immediate 145
initialize TTY 142
input unit 27
input/output 43
instruction 27
integrated circuit 19
interface 43
interface circuits 43
interrupt request 111

jump 35
jump and store 121
jump dollar 148

keyboard 126

label 163
large scale integration 13
latch 78
least significant bit 73
LF 144
line feed 144
loop 68
LSB 26,73

mark 133
memory acknowledge 111
memory reference instructions 62
microinstruction 27
microprogram 27
mnemonic 65
MUX 108

OEM 54
omkopplare 128
option 136
original equipment manufacturer 54
output unit 27

parity check 133
PC 34
peripherals 43
power fail 123
power supply 122
preconditioning 43
program counter 34

random access memory 38,87
random logic 103
read only memory 38,83
read only storage 83
real time clock 140
recall 30
record 45
register file 88
return 35
RTC 140
RUN ENABLE MODE 69
RUN MODE 69

scientific calculator 28
semiconductor 19
seven segment display 22
source program 164
space 133
STEP MODE 69
STOP MODE 69
store 30

tape punch 126
tape reader 126
timing unit 103
typing unit 126

universal asynchronous receiver
transmitter 138

volatile 42
wired logic 103
word length 27
write 43

För utbildning både inom industrin och i skolan har man vid olika tillfällen efterlyst en »datorbok» som på ett enkelt men ändå inte ytligt sätt ger en brygga mellan elektroniken i datorn och datorns programmering. Den här boken är ett försök att fylla detta behov.

Boken inleds med en beskrivning av byggblocken i en räknedosa och en programmerbar kalkylator. Styrenhetens arbetssätt och mikroprogrammering är fundamentala för förståelsen av en dators funktion. Boken ger därför en konkret beskrivning både av de elektroniska kretsarna och av mikroprogrammets uppbyggnad i en modern minidator. Efter genomgången av styrenhetens funktion följer på ett naturligt sätt en beskrivning av maskinkoden och assemblerprogrammets uppbyggnad. Därmed är bryggan mellan maskinvara och programvara sluten.

Boken förutsätter att läsaren har elementära kunskaper om transistorns funktion (eller skaffar sig dem via de i boken givna lättillgängliga litteraturkällorna).

FOLIANT
UMEÅ KOMMUNS BIBLIOTEK
Stadsbiblioteket
MAG
2 284 305 2480 40